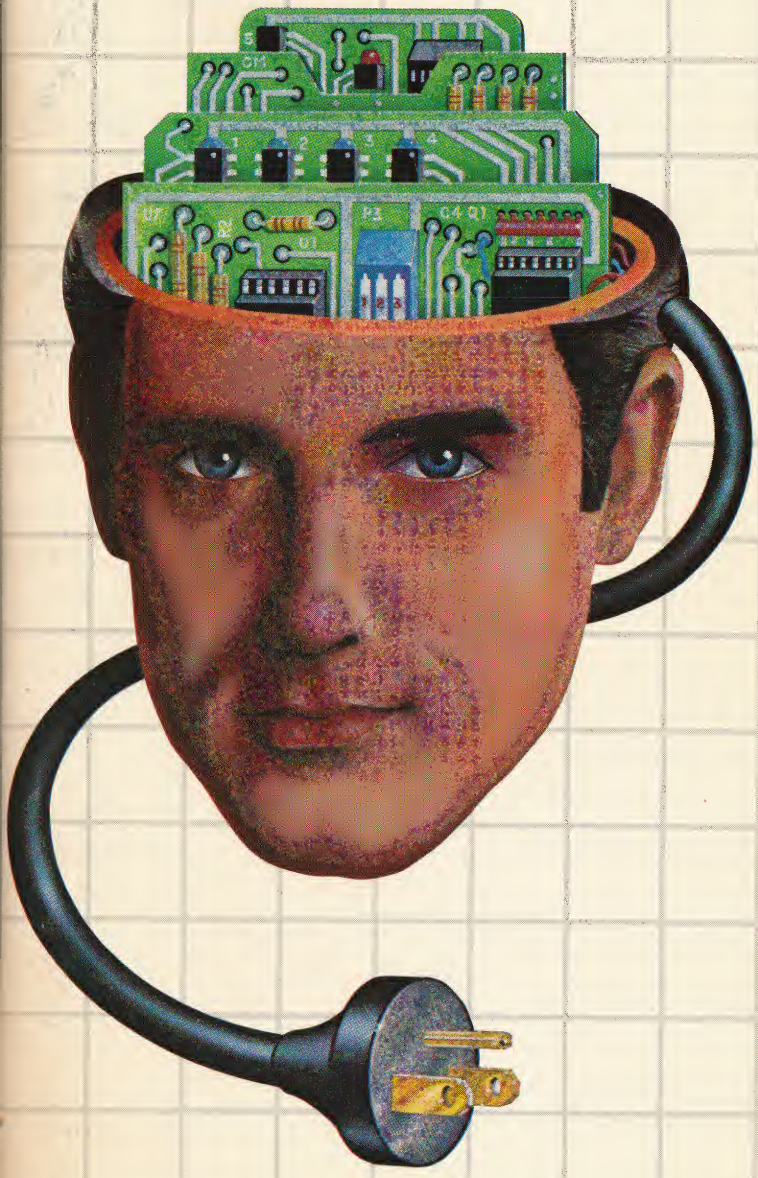
$2.95
in U.S.A.

UNDERSTANDING LISP

A Concise Introduction to the Language of Artificial Intelligence

An Alfred Handy Guide



By Paul Y. Gloess

UNDERSTANDING LISP

by Paul Y. Gloess

AN ALFRED HANDY GUIDE

Computer Series Editor:
George Ledin Jr.

ALFRED PUBLISHING CO., INC.
SHERMAN OAKS, CA 91403

Editorial Supervision: Joseph Cellini

Cover Design: Paula Bingham Goldstein

Production Management: Michael Bass & Associates

Alfred Publishing Co.,Inc.
15335 Morrision Street
P.O. Box 5964
Sherman Oaks, CA 91413

Library of Congress Cataloging in Publication Data

Gloess, Paul Y.
Understanding LISP.

(An Alfred handy guide)
1. LISP (Computer program language) I. Title.
QA76.73.L23G57 1982 001.64'24 82-18464
ISBN 0-88284-219-6

CONTENTS

1. THE ORIGINALITY OF LISP

LISP was invented by John McCarthy two decades ago. It is now the most widely used programming language in the Artificial Intelligence community and is gaining acceptance in other domains such as system design (e.g., text processing). Unlike languages like FORTRAN, which were designed for numerical calculations (payroll, finances, mechanical engineering), LISP is especially good at manipulating symbolic expressions. Its expressive power and originality are based on the following properties:

- LISP has essentially one *data-type*, called *S-expressions*, (short for "Symbolic expression"), which allows representation of structured information in a natural way. A typical LISP object is the *list* (COLOR GREEN) of the two *atoms* COLOR and GREEN. A slightly more complex object is the list ((COLOR GREEN) HELLO), whose first element is the list (COLOR GREEN) and second element the atom HELLO. Elements are easily accessed or combined by means of three basic functions–the *selectors* CAR and CDR, and the *constructor* CONS.

- LISP programs are themselves lists (a kind of S-expression). As a result, LISP programs can easily be used to manipulate or build LISP programs!

- LISP is highly interactive. LISP *forms* or programs are immediately executable without preprocessing by the so-called LISP interpreter. (Other languages usually require compiling before executing.)

- "Pure" LISP requires only five primitive functions (CAR, CDR, CONS, ATOM–which decides whether a given object is an atom–and EQ–which tests the equality of the addresses of two objects) and two forms (COND for conditional expressions and LAMBDA for the definition of functions). Everything else, including the LISP interpreter, can be built from these.

- The LISP interpreter is readily available as a function called EVAL, which can be invoked from user-defined programs. A user program can thus build another program and execute it.

- LISP is a *recursive* language, that is, a program can be defined in terms of itself. This is very important because many problems have a recursive nature.

- Memory management is automatic. The programmer does not need to declare the size of the objects that will be built during the execution. Objects that are no longer in use are automatically erased and recycled by the garbage collector.

- *Property lists* provide a means of attaching information to names and accessing that information in a selective way. Data bases can be manipulated with ease using property lists.

More than a language, LISP is a programming environment. Good LISP systems usually come with packages (written in LISP) that greatly facilitate the task of programming: powerful debuggers, sophisticated editors and program analyzers, programmer's assistants, and measurement

tools. If all these remarkable packages now exist in LISP and are poor or nonexistent in other languages, it is because the principles described above made them relatively easy to build! A good environment is essential to software design and production, and this is a major reason for the growing success of LISP.

Several good LISP systems are now available on home micro-computers: the LISP Company "(T . (L . C))", is selling one for the Radio Shack TRS-80; VLISP (now called "Le LISP") is a popular French LISP that runs on TRS-80s and various DEC PDP machines. Such systems are advertised in *BYTE* and other magazines about small computers and their uses.

LISP is an evolving language with several dialects. The two most widely used are MACLISP and INTERLISP. MIT and the Xerox Palo Alto Research Center are among the leading developers of LISP. They have built LISP-oriented machines that are the most advanced tools ever designed for software or hardware development. In these machines every part of the system, from the micro-coded interpreter to the sophisticated scheduler or graphic routines, is written in LISP and accessible to the user in a unified way!

Figure 1-1 shows the LISP machine developed by LMI in collaboration with MIT. A powerful *window-system* allows the user to control several tasks running in parallel: the screen looks like a desk covered by sheets of paper, one sheet per task. These sheets can be moved and can overlap; they can display animated drawings, computations, programs, text, mail, etc. A little mouse marks the screen location.

FIGURE 1-1. The LMI LISP Machine, Courtesy LMI

2. THE MAGIC OF RECURSION

A *recursive function* (or program) is one that is defined in terms of itself.

Modern programming languages such as ALGOL, PL/I, APL, Pascal, and SNOBOL support recursion. LISP does; FORTRAN does not. As a result, many problems that are very hard to solve in FORTRAN become surprisingly easy to solve in LISP or other recursive languages. A student who has produced, with great effort, hundreds of lines of BASIC or FORTRAN that failed to solve a problem, will stare in disbelief at four lines of LISP achieving the desired effect!

In LISP, recursion is a possibility offered to the programmer, not an obligation. By never using it, one can easily write "horrible LISP programs," just as horrible as the most horrible FORTRAN programs. Inappropriate use of this tool is also possible.

"Thinking recursive" seems to be natural for some people and very difficult for others. It is generally easier if you have never been exposed to a nonrecursive programming language. Otherwise, the main difficulty is to convince yourself that simplicity can work.

2.1 CIRCULARITY AND RECURSION

One might think that defining something in terms of itself is nonsense. Indeed, no one would be happy with the following definition of a bicycle:

> A **bicycle** is an object you can buy in a **bike shop**, and a **bike shop** is a shop where **bicycles** are sold.

The above statement is clearly circular: it does not tell us what a bicycle is unless we know what a bike shop is—and to know that, we would have to know what a bicycle is!

Let us consider another example, a definition of the word **ancestors:**

> Adam and Eve have no **ancestors.**
> The **ancestors** of someone else comprise his (her) mother and father, his (her) mother's **ancestors** and his (her) father's **ancestors.**

This definition may seem circular at first glance: in the second part, the word ancestors is recursively used twice. The ancestors of a person are defined in terms of the ancestors of the mother and those of the father of this person. In fact, this definition turns out to be perfectly sound and applicable to practical situations (assuming that public records always tell us who is the mother and who is the father of anyone, except Adam and Eve). To illustrate the truth of our claim, we will apply it to the computation of John Smith's ancestors, whose (partial) genealogical tree is shown in Figure 2-1.

A first application of the definition tells us that the **ancestors of John Smith** are:

- Lucy Parker (his mother)
- Steven Smith (his father)
- the **ancestors of Lucy Parker**
- the **ancestors of Steven Smith**

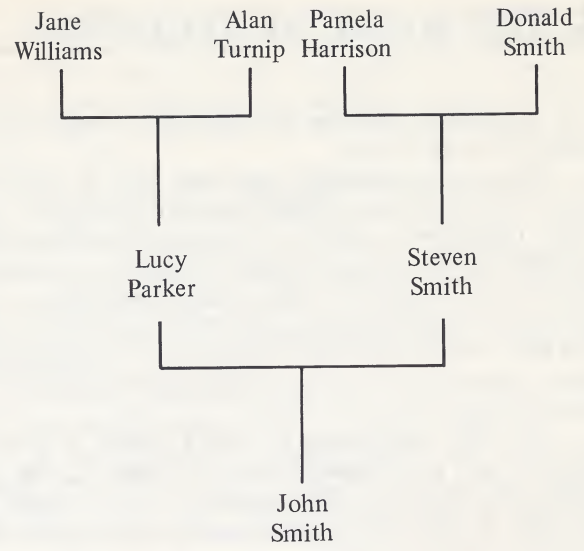


FIGURE 2-1. A Partial Chart of John Smith's Genealogy

A second application is now necessary to determine the **ancestors of Lucy Parker.** They are:

- Jane Williams (her mother)
- Alan Turnip (her father)
- the **ancestors of Jane Williams**
- the **ancestors of Alan Turnip**

From a third application of the definition, the **ancestors of Steven Smith** are:

- Pamela Harrison (his mother)
- Donald Smith (his father)
- the **ancestors of Pamela Harrison**
- the **ancestors of Donald Smith**

By combining the above results, we obtain a list of **John Smith's ancestors**:

- Lucy Parker
- Steven Smith
- Jane Williams and **her ancestors**
- Alan Turnip and **his ancestors**
- Pamela Harrison and **her ancestors**
- Donald Smith and **his ancestors**

We seem to generate more work than we complete: after three applications of the definition, we now have to deal with the ancestors of four different persons. Shall we ever reach the end of it? Yes, if everybody comes from Adam and Eve!

It should be clear from this example that the concept of recursion is independent of the method of computation and the programming language used.

The next story, the "Towers of Hanoï," illustrates the power of recursion and introduces the LISP way of dealing with it.

2.2 THE TRUE STORY OF THE TOWERS OF HANOÏ

Long, long ago, Zeus decided to punish Apollo for spending too much time among the women. He called him and said:

"Thou shalt go to Hanoï and find three spikes: on the left one I have stacked nine golden disks, in order of decreasing size. Thou wilt move them onto the right spike in the same order."

"No problem!" said Apollo.

"Do not interrupt Zeus! Thou shalt move one disk at a time. Never shalt thou put a disk on top of a smaller one."

"What use is thy middle spike?"

"Fool! Without that spike thy task would be impossible, for thou must stack each disk onto some spike before unstacking another one. If thou makest one mistake, thou shalt start again with the nine disks on the left spike!"

Three thousand years later, the golden disks were almost worn out and Apollo had not succeeded. Zeus paid him a visit: "Why dost thou not use LISP? Here is a microcomputer, with color graphics display, and a LISP diskette."

2.3 A FIVE-LINE RECURSIVE SOLUTION

Here is a remarkably simple LISP program, called **HANOI**, that assumes entirely by itself the task of deciding at each step which disk should be moved and where:

```
(DEFINE 'HANOI' (N SOURCE INT DEST)
        '(COND [(EQUAL N 0 NIL)
                [T (HANOI (SUB1 N) SOURCE DEST INT)]
                   (MOVE_ONE_DISK)
                   (HANOI (SUB1 N) INT SOURCE DEST) ()])
```

The reader should not try to understand this program but just notice that it is recursive, i.e. the definition of HANOI includes the two shaded lines which include HANOI and are the heart of the program. That is, every time the program comes to one of these shaded lines, it will substitute for HANOI the entire definition of HANOI, which in turn includes the two recursive calls, and so on. (HANOI solves the problem not only for nine disks–which takes $2^9 - 1 = 511$ basic moves–but for any number of disks.)

A complete understanding of HANOI will be possible only after reading Chapter 5. Section 6.7 thoroughly explains MOVE_ONE_DISK. The purpose of the next two sections is to present the essential idea of HANOI and give a foretaste of LISP mechanics.

2.4 THE NEAT RECURSIVE IDEA OF HANOI

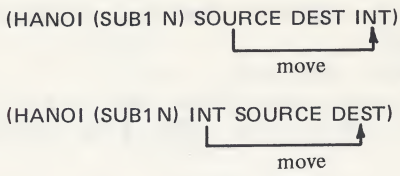
On the first line of the HANOI program we see the list:

```
(N SOURCE INT DEST)
```

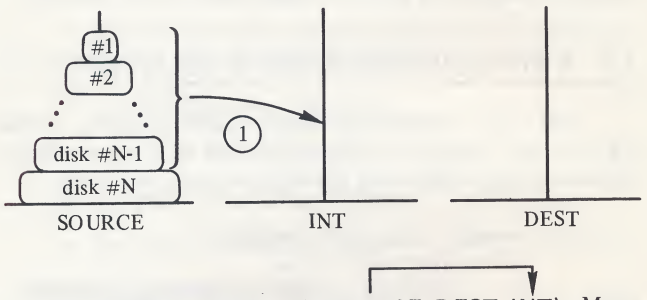
This is what we shall call the *list of parameters* of HANOI. N represents the number of disks to be moved; SOURCE represents the SOURCE tower from which the disks are removed; INT is the INTermediate tower, used as an auxiliary one during the transfer of disks; finally, DEST is the

DESTination tower: when it holds the N disks, the problem is solved.

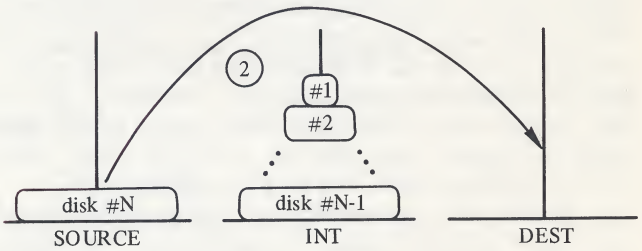
The order of the parameters in the list is significant: SOURCE is the second parameter and DEST is the fourth one. In the recursive calls, the disks move from the tower specified by the second parameter to the tower specified by the fourth one:



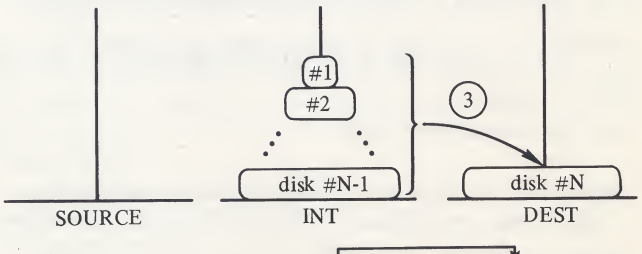
The neat recursive idea of HANOI is to assume that we know how to move N-1 disks from one tower to another one without violating Zeus's rule, and deduce from there a method for moving N disks that still obeys Zeus's rule. This method is summarized in the diagrams below:



1. (HANOI (SUB1 N) SOURCE DEST INT): Move the N-1 topmost disks of SOURCE to INT. (This is the first recursive call.)



2. (MOVE_ONE_DISK): Move the (topmost) disk of SOURCE onto the top of DEST.



3. (HANOI (SUB1 N) INT SOURCE DEST): Move the N-1 topmost disks of INT onto the top of DEST (second recursive call).

It should be clear that after Steps 1, 2, 3 have been completed, the N disks are properly stacked on the DESTination tower. Furthermore, since steps 1 and 3 only manipulate disks #1 through #N-1—all smaller than disk #N—disk #N cannot cause any violation of Zeus's rule during the execution of these steps. In other words, the presence of disk #N does not interfere with the correct execution of steps 1 and 3. Step 2 is obviously correct, since it moves the only disk of SOURCE to the empty tower DEST. Therefore, assuming the correctness of HANOI for moving a stack of N-1 disks (with $N \geq 1$), we have proved the correctness of HANOI for moving a stack of N disks. Equivalently, *if HANOI is correct for N disks, $N \geq 0$, it is correct for N + 1 disks.*

For an empty stack of disks—i.e., N = 0—HANOI does nothing, which is trivially correct! *HANOI works correctly for N = 0.* Hence, HANOI is correct for 0 + 1 = 1 disk. Being correct for 1 disk, it is correct for 1 + 1 = 2 disks, and so on . . . *HANOI is correct for any number of disks!*

2.5 HOW HANOI TRANSFERS A STACK OF THREE DISKS

The transfer of three disks from tower A to tower C by the HANOI program is shown in Figure 2-2: seven moves are performed by MOVE_ONE_DISK. The initial and the recursive calls to HANOI are emphasized by vertical square brackets that delimit each call. Note the values of the arguments: e.g., shaded HANOI (2 A C B) meaning N = 2, SOURCE = A, INT = C, and DEST = B—these are the values accessible to the shaded MOVE_ONE_DISK, which explains the transfer of one disk from SOURCE = A to DEST = B. In general, the values accessible to MOVE_ONE_DISK are those at the entrance of the smallest embedded HANOI call.

See Figure 2-2 on the following page.

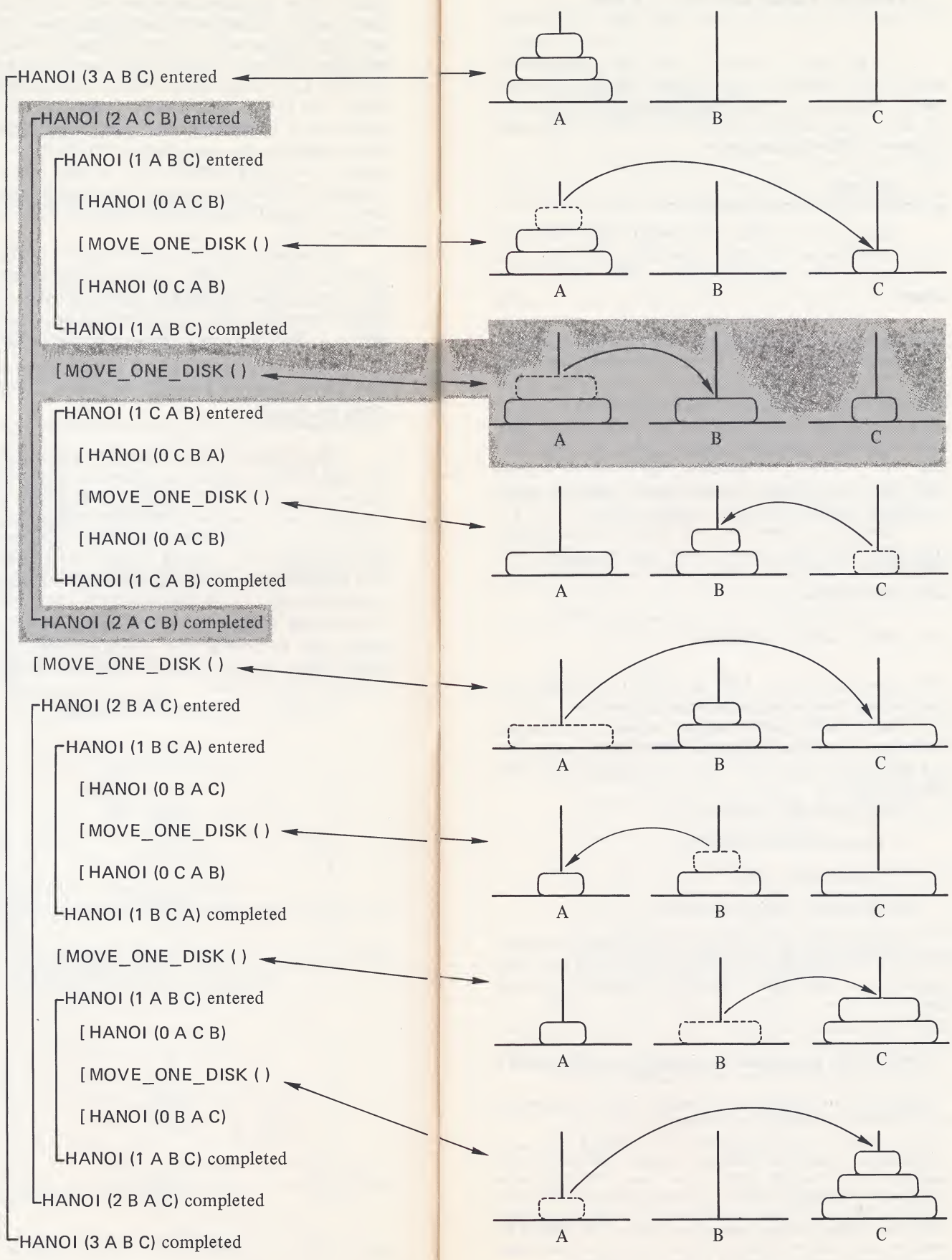


FIGURE 2-2. How HANOI Proceeds for Three Disks.

3. S-EXPRESSIONS ARE THE ONLY LISP DATA TYPE

In LISP, there is only one data type, *S-expressions*, which include objects of two varieties, *atoms* and *dotted pairs*.[1] This data type is extremely simple, yet very powerful. Most of the structures that we can think of can easily be represented in this data type.

3.1 ATOMS

These are the smallest LISP objects. They cannot be broken into pieces.[2] An atom looks like a name. It is made of characters, the first one being a letter. Here are six atoms:

```
HELLO       WELCOME      SEE_YOU_LATER
GOODBYE     X            MYCADILLAC
```

The only restriction to the length of an atom, if any, is the size of a line. Every LISP system has its own conventions regarding characters allowed after the first one. SEE_YOU_LATER, for instance, may be invalid in some LISP dialects (the blank character cannot normally belong to an atom because it is used to separate atoms).

Numbers are atoms, but they do not look like names. (The kind of atoms shown above may be called *literal atoms* if the distinction matters.) Here are some numbers, in decimal notation:

```
100     -150     138.275
```

The primary purpose of LISP is not that of dealing with numbers, but it is occasionally useful to do so.[3]

Many LISP systems support strings of characters (sometimes considered exactly like literal atoms, sometimes as a distinct type of atom).[4] Their sole purpose is to allow fancier output.

Atoms come in three flavors:

- literal atoms (e.g., HELLO)
- numbers (e.g., 200)
- strings (e.g., "This is a string!")

Literal atoms have very special properties when they are used in LISP programs; for the purpose of the basic data operations we study next, however, no distinction is necessary.

---

[1] Powerful LISP systems such as INTERLISP actually support a few others, e.g., arrays. However, they are seldom used.

[2] Again, this is not strictly true, since many LISP dialects provide ways of breaking atoms into their constituent characters.

[3] It is generally thought that LISP is intrinsically bad for numeric computations. MACLISP's numeric compiler provides a good counter-example. The LMI LISP machine beats FORTRAN on its own ground: it supports "infinite precision" arithmetic! Thus, the factorial of 300 is evaluated immediately–the result takes several thousands of digits, but the program is one line of LISP. FORTRAN could not make this calculation without a lengthy special-purpose program, since numbers are limited to 8 bytes of memory.

[4] Not all LISP systems that support strings treat them as atoms.

3.2 DOTTED PAIRS

From atoms one can build more complex objects called *dotted pairs*. Here are some dotted pairs, made of the atoms from section 3.1:

```
(HELLO . X)
(GOODBYE . MYCADILLAC)
(WELCOME . WELCOME)
```

Note the use of a period and parentheses to form a dotted pair. This operation of building dotted pairs is called *CONSing* (for a reason that will be clear later on).

It is possible to **CONS** numbers together, or a number and a literal atom, as in:

```
(100 . -150)      (X . 138.25)
```

(The "." used to represent the CONS should be surrounded with blanks, unlike the decimal point in "138.25".)

It is also possible to CONS dotted pairs (or atoms) with dotted pairs (or atoms). The resulting objects are also called dotted pairs. For example, the CONS of the atom HELLO and the dotted pair (HELLO . X) is the dotted pair:

```
(HELLO . (HELLO . X))
```

and the CONS of the dotted pairs (GOODBYE . MYCADILLAC) and (WELCOME . WELCOME) is the dotted pair:

```
((GOODBYE . MYCADILLAC) . (WELCOME . WELCOME))
```

There is no limit to the complexity of dotted pairs, for the CONSing operation can be repeated at will.

It is important to understand that an object such as

```
(HELLO . (HELLO . X))
```

is just a representation of an abstract object called a dotted pair. In this example, the first "." (from the left) represents the main CONS. This dotted pair has two components:

- The left part–which starts after the leftmost opening parenthesis and ends before the main (here, the first) "."–represents the **CAR** of the dotted pair. Here this CAR is the atom HELLO.
- The right part, (HELLO . X)–which starts after the main "." and ends before the rightmost ")"–represents the **CDR** (pronounced "coo-dra") of the dotted pair. This CDR of the original dotted pair is itself a dotted pair whose CAR is the atom HELLO and CDR is the atom X.

When a dotted pair is complex, it becomes difficult to see its structure just by looking at its representation. Identifying the main "." requires careful matching of each "(" with the corresponding ")" (shown in Figure 3-1).

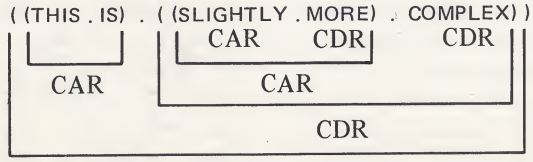


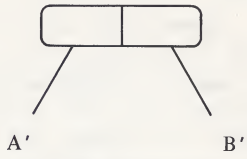
FIGURE 3-1. Identifying the Structure of a Dotted Pair

We shall now study another very useful representation of S-expressions.

3.3 BINARY TREES TO REPRESENT S-EXPRESSIONS

The previous representation, a linear text with parentheses and periods, is a convenient means of interaction with the computer. *Binary trees* are actually much closer to the internal representation used by the computer, and they display the structure of dotted pairs more clearly.

- The binary tree representation of an atom is the atom itself.
- The binary tree representation of a dotted pair, say (A . B), where A (the CAR) and B (the CDR) represent an atom or a dotted pair, is



where A′ is the binary tree representation of A, and B′ the binary tree representation of B.

Figure 3-2 shows the binary representation of one atom and two dotted pairs.

The  is called a *list cell* or *cell*. Each cell corresponds to a "." (or CONS) in the previous notation. The topmost cell in the tree corresponds to the main CONS in the dotted pair. Two branches issue from each cell; the left one corresponds to the CAR and the right one to the CDR.

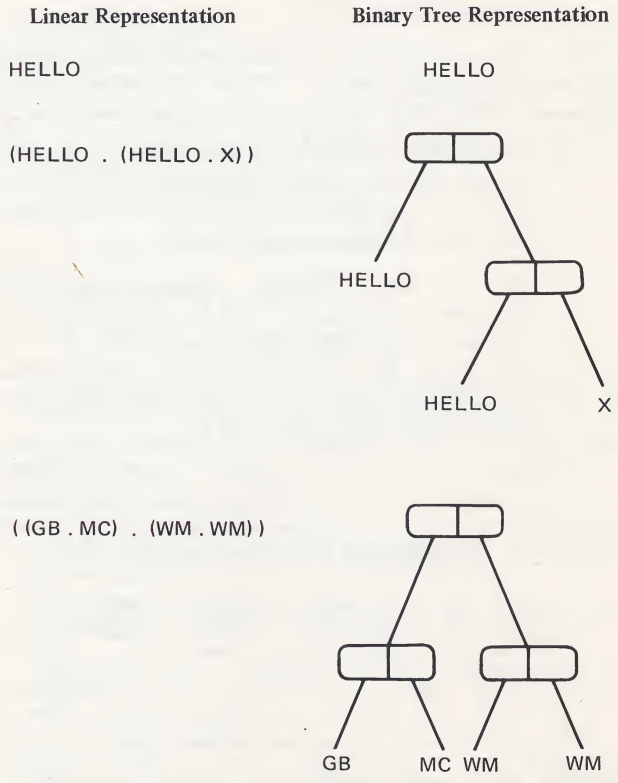


FIGURE 3-2. Binary Tree Representation of S-Expressions

3.4 MACHINE REPRESENTATION OF DOTTED PAIRS

In the memory of a computer, each list cell occupies one double word of memory.[5] The left half contains a number—the memory address of the topmost cell of the CAR (or that of an atom if the CAR is an atom); the right half contains the address of the CDR. Thus it suffices to know the address of the topmost cell of the tree to know the whole tree. Figure 3-3 illustrates this principle (with addresses chosen randomly): the topmost cell is at address 12300.

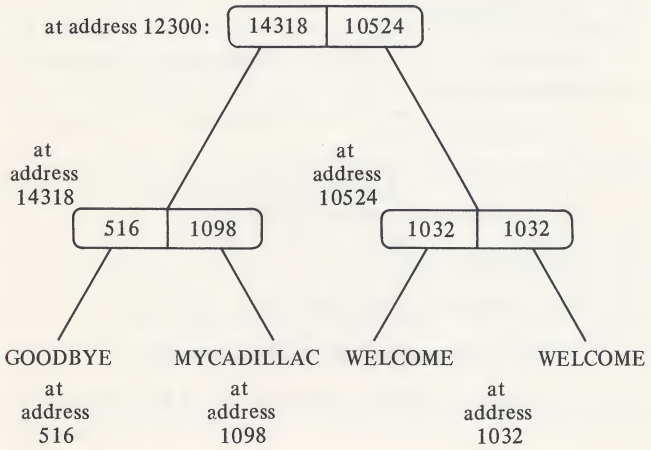


FIGURE 3-3. Machine Representation of "((GOODBYE . MYCADILLAC) . (WELCOME . WELCOME))"

3.5 THE ABSTRACT DATA TYPE OF S-EXPRESSIONS

We have already introduced three of the basic functions involved in this abstract data type:

CONS: a function of two arguments, which may be atoms or dotted pairs; the result is a dotted pair

CAR: a function of one argument, which should be a dotted pair; the result is a dotted pair or an atom

CDR: a function of one argument, which should be a dotted pair; the result is a dotted pair or an atom.

CONS is called a *constructor* because it is used to build up dotted pairs from smaller components. CAR and CDR are called *selectors* because they can select the components of a dotted pair.

Another useful function in this context is **DTPR**. DTPR tells us whether an object is a dotted pair: when DTPR is applied to an object x, it returns the value "true" if x is a dotted pair, "false" otherwise. One can then define the function **ATOM** to return exactly the opposite value. Intuitively, in a world, like that of pure LISPs, of atoms and dotted pairs, ATOM recognizes the atoms and DTPR recognizes the dotted pairs. The traditional name for DTPR is

---

5 There are other implementations, but this one is typical of most LISP systems.

LISTP. However, we prefer DTPR here, because LISTP is a source of confusion: dotted pairs and lists (see section 3.7) are not the same!

DTPR (or LISTP) and ATOM are called *recognizers*. They are also called *predicates*, because they return logical (true or false) values.

Up to now, we have presented the data type of atoms and dotted pairs in an intuitive way. This approach is satisfactory to most people, but a precise definition is possible, by means of axioms that relate these functions to each other. Here are some important axioms and their explanations: the reader should recognize some obvious facts in agreement with the intuitive approach.

DTPR(CONS(x, y))

The CONS of two objects x and y is a dotted pair. (It is not an atom.)

CAR(CONS(x, y)) = x

The CAR of the CONS of x and y is x. ("=" is to be understood in its mathematical sense.)

CDR(CONS(x, y)) = y

The CDR of the CONS of x and y is y.

DTPR(x) ⇒ x = CONS(CAR(x), CDR(CDR(x))

A dotted pair is the CONS of its CAR and its CDR.

3.6 A SPECIAL ATOM CALLED NIL OR ( )

Nothingness is important in many domains. One of the greatest inventions in mathematics was the number 0. **NIL** is to S-expressions what 0 is to numbers. Although NIL is an atom, it is sometimes called the *empty list* and sometimes written ( ), as it plays an important role in the definition of *lists*.

3.7 LISTS AND PROPER LISTS ARE SPECIAL S-EXPRESSIONS

We have now completely introduced the fundamental LISP data type of S-expressions. *Lists* are not a new kind of object: they are a restricted kind of S-expression. The class of *proper lists* is even more restricted but is sufficient for all applications.

- Roughly speaking, *any problem that can be solved by means of S-expressions can be solved in a similar way by means of proper lists.*

As a matter of fact, it is possible to learn about proper lists without knowing about S-expressions! Whereas dotted pairs are composed of two elements, proper lists can have any number of elements—even zero, as in the empty list ( ). Here are some simple proper lists (note the absence of periods in this notation):

( )	(HELLO)	(WELCOME BACK)
(HELLO HELLO BYE)	(THIS IS NOT NIL)	NIL

The first and the last are the same empty list. The other four lists have, respectively, one, two, three, and four elements, which are atoms. Note that (HELLO HELLO BYE) is considered to have three elements even though the first two are identical. The order of the elements is important:

(WELCOME BACK) is not the same as (BACK WELCOME).

Proper lists can have proper lists as their elements, for example:

(NIL)	((HELLO HELLO) BYE)	((THIS IS) (NOT NIL))
((NIL))	(WELCOME (BACK))	(THIS (IS NOT) NIL)

NIL, (NIL) and ((NIL)) are three distinct proper lists. ((THIS IS) (NOT NIL)) has two elements, the proper lists (THIS IS) and (NOT NIL), whereas (THIS (IS NOT) NIL) has three elements: the first one, THIS, is an atom, the second one is the proper list (IS NOT), and the third one is the atom (also a proper list) NIL.

Lists are like proper lists except that their elements are unrestricted: they can be any S-expressions. Thus, (A (B . C) C) is a list, but not a proper list. (This distinction will soon be made clear.)

We will now develop a rigorous definition of proper lists and lists in terms of S-expressions because:

1. S-expressions are essential to a deep understanding of lists and list manipulation.
2. Even if one decided to ignore S-expressions and deal with proper lists exclusively, dotted pairs that are nonlists could easily be produced inadvertently from atoms and proper lists! The results would then be very confusing for a beginner.

Section 3.8 will combine the intuitive with the formal approach.

Definition 1. A *list* is either the atom NIL or a dotted pair whose CAR is an S-expression and whose CDR is a list.

Definition 2. A *proper list* (p-list) is either the atom NIL or a dotted pair whose CAR is an atom and whose CDR is a proper list, or a dotted pair whose CAR is a proper list and whose CDR is a proper list.

These are typical recursive definitions. They should make some sense to the reader who has studied Chapter 2. However, the best way to understand Definition 2, for instance, is to build up some proper lists, from simple to complex.

NIL	by definition
(IT . NIL)	CONS of atom IT and p-list NIL
(IS . (IT . NIL))	CONS of atom IS and above p-list
(THIS . (IS . (IT . NIL)))	CONS of atom THIS and above p-list

Here is how to build up ((THIS . (IS . NIL)) . (IT . NIL)):

(IT . NIL)	CONS of atom IT and p-list NIL
(IS . NIL)	CONS of atom IS and p-list NIL
(THIS . (IS . NIL))	CONS of atom THIS and above p-list
((THIS . (IS . NIL)) . (IT . NIL))	CONS of above p-list and first p-list (IT . NIL)

The following are S-expressions that are not p-lists:

```
GOODBYE                    (BYE . BYE)
((BYE . BYE) . (HI . NIL))  (NIL . (GOOD . MORNING))
```

For example, the last is not a p-list because MORNING is not a p-list, so (GOOD . MORNING) is not a p-list.

A dotted pair is a p-list if only the atom NIL appears immediately to the right of a period. (The corresponding binary tree has all its *right leaves* equal to NIL.)

3.8 LIST NOTATION SAVES PARENTHESES AND PERIODS

The usual linear representation of dotted pairs can sometimes be simplified by the following transformations:

1. Whenever a "." is immediately followed by NIL (or ( )), erase the "." and the NIL (resp. ( )).
2. Whenever a "." is immediately followed by a "(", erase the ".", the "(", and the corresponding closing ")".

The resulting form is called *list notation*. Note that the first of these transformations is just an instance of the second one, provided that NIL is always written ( ). Sometimes these transformations eliminate all periods.

P-lists and atoms are the only S-expressions whose list form contains no period.

Here is an example of step-by-step transformation of a dotted pair. Each pair of parentheses and period to be erased in the next line appears in boldface.

```
((HI . NIL) . (HOW . (ARE . YOU)))      original form
((HI) . (HOW . (ARE . YOU)))            by first rule
((HI) . (HOW ARE . YOU))                by second rule
((HI) HOW ARE . YOU)                    by second rule
```

This dotted pair is not a p-list, because the last period cannot be removed.

We shall now call the usual linear form of S-expressions (that is, the first form we introduced) *dot form*. It is easy to see that *it is always possible to go from list form to dot form*, by applying the following rules:

1. Replace ( ) with NIL.
2. If a list has one or more elements, place a "." to the right of the first element and enclose the remaining elements in a pair of parentheses. (Use an empty pair of parentheses if there is no remaining element.)

For example, the list ((WATCH OUT) (FOR) WAVES) can be put step by step into dot form as follows (newly introduced parentheses, period, or NIL in boldface):

```
((WATCH OUT) (FOR) WAVES)                          initially
((WATCH OUT) . ((FOR) WAVES))                      by rule 2
((WATCH OUT) . ((FOR) . (WAVES)))                  by rule 2
((WATCH OUT) . ((FOR) . (WAVES . ())))             by rule 2
((WATCH OUT) . ((FOR) . (WAVES . NIL)))            by rule 1
((WATCH . (OUT)) . ((FOR) . (WAVES . NIL)))        by rule 2
((WATCH . (OUT . ())) . ((FOR) . (WAVES . NIL)))   by rule 2
((WATCH . (OUT . NIL)) . ((FOR) . (WAVES . NIL)))  by rule 1
((WATCH . (OUT . NIL)) . ((FOR . ()) . (WAVES . NIL)))  by rule 2
((WATCH . (OUT . NIL)) . ((FOR . NIL) . (WAVES . NIL)))  by rule 1
```


Note the economy of parentheses and periods in the list form!

It is easy to see that CAR, CDR, and CONS have the following interpretation when applied to an S-expression x in list form:

CAR of x: first element of list x

CDR of x: list of all elements of x except the first (i.e., the *rest* of x)

CONS of e and x: the result of inserting e between the opening "(" of the list x and its first element (e is any S-expression, in any form)

For instance, the CAR of (A B C) is A and its CDR is (B C); the CONS of A and (B C) is (A B C). This is in perfect agreement with the fact that (A B C) is just an abbreviation for (A . (B . (C . NIL))) and (B C) for (B . (C . NIL)).

Figure 3-4 shows the possible results of a CONS, in both list and dot notations. Figure 3-5 does the same for CAR and CDR. In these tables, <S-expr>, $\overline{\text{<S-expr>}_1, \ldots, \text{<S-expr>}_n}$ designate dot forms of S-expressions; $\overline{\text{<S-expr>}, \ldots}$ designate their list-form counterparts; <atom> stands for any atom.

$(\text{<S-expr>}_1 \ \ldots \ \text{<S-expr>}_n)$ is a list of n S-expressions.

CONS: First Argument \ Second Argument	NIL	<atom>	(<S-expr>$_1$. (. . . . (<S-expr>$_n$. NIL) . . .))	
		HI	($\overline{\text{<S-expr>}}_1$ . . . $\overline{\text{<S-expr>}}_n$)	List Form
<S-expr>	(<S-expr> . NIL)	(<S-expr> . <atom>)	(<S-expr> . (<S-expr>$_1$. (. . . . (<S-expr>$_n$. NIL) . . .)))	Dot Form
SIMPLE	(SIMPLE . NIL)	(SIMPLE . HI)	(SIMPLE . (FIRST . (LAST . NIL)))	
$\overline{\text{<S-expr>}}$	($\overline{\text{<S-expr>}}$)	($\overline{\text{<S-expr>}}$. <atom>)	($\overline{\text{<S-expr>}}$ $\overline{\text{<S-expr>}}_1$ . . . $\overline{\text{<S-expr>}}_n$)	List Form
SIMPLE	(SIMPLE)	(SIMPLE . HI)	(SIMPLE FIRST LAST)	

FIGURE 3-4. Effect of CONS in List and Dot Notations. White Areas Show the General Case; Shaded Boxes Show an Example

Function \ Argument	NIL	<atom>	(<S-expr>)	(<S-expr>$_1$ . . . . <S-expr>$_n$)	
			(ONE)	(ONE TWO THREE)	
CAR	***	***	<S-expr>	<S-expr>$_1$	
			ONE	ONE	
CDR	***	***	NIL	(<S-expr>$_2$. (. . . . (<S-expr>$_n$. NIL) . . .))	Dot Form
				(TWO . (THREE . NIL))	
				($\overline{\text{<S-expr>}}_2$ . . . $\overline{\text{<S-expr>}}_n$)	List Form
				(TWO THREE)	

***: "undefined" or "implementation dependent."

FIGURE 3-5. Effect of CAR and CDR. White Areas Show the General Case; Shaded Areas Show an Example

4. TALKING TO A LISP SYSTEM

Because LISP systems are highly interactive, users sitting at a terminal are not limited to typing programs and then running these programs with some data, as they would have to do with other programming languages. The user can type a piece of program, from the simplest to the most complex: If this piece of LISP is "correct" the value of this piece will immediately appear as an answer.

A piece of LISP code is called a *LISP form* or *form.*

Be polite and type only correct forms to your LISP system, or it will send you a warning, instead of the expected value.

Please **always** close all your parentheses, or you will run out of patience before LISP does! (Nothing happens until you have typed a complete form.)

The simplicity of its grammar makes it easy to write using LISP. In fact, in learning about lists the reader has already learned most of the syntax of LISP.

4.1 BLANKS AND END-OF-LINE

Blanks are used to separate atoms from each other or atoms from periods. One blank is enough, but additional blanks may be used for clarity. The *end-of-line*, usually signaled by the RETURN key, does not mean the end of the form; instead, it has the same logical function as the blank. The end of the form is reached when the first opening parenthesis has been closed. Some terminals transmit data line by line to the computer: in that case, an end-of-line is requested after the last ")" to transmit the last line. When the form is atomic, a space or an end-of-line is required to signal the end of the atom.

4.2 LISP FORMS

LISP forms are S-expressions (usually proper lists or atoms) that can be evaluated in some environment. The **value** of a form depends on the environment at the time of the evaluation. The user has the means of changing the environment, and the environment changes during the process of evaluation. Thus the same form can have one value in a certain environment and be undefined or have a different value in another environment.

What is an **environment**? For the time being it will suffice to know that LISP maintains a table of all the atoms ever introduced. Every atom may be *bound to* a value, which can be any S-expression, or *unbound*. The *table of atoms*, together with the particular bindings of the atoms, constitutes a part of an environment.

Definition 1: A *LISP form* is either an atom or a list of the form: $(f\ \mathrm{arg}_1\ \ldots\ \mathrm{arg}_n)$

where

f, the first element of the list, is an atom that names an existing function that takes exactly n arguments (it is said to be of *arity* n);[1]

the elements $\mathrm{arg}_1, \ldots, \mathrm{arg}_n$ are *LISP forms.*

[1] We are simplifying the situation somewhat; we will see that f may also be a LAMBDA-expression (see section 5.10).

Once again we have formulated a recursive definition. We have also used the ". . ." notation, which may confuse some readers. The ". . ." is not part of the form and should not be typed by a user! It is just standard mathematical notation indicating a variable number of elements. In each specific instance, of course, this number is fixed: recall, for example, that CONS has arity 2 and should be followed by exactly two forms, whereas CAR, CDR, ATOM, and LISTP have arity 1 and should be followed by only one form.

Here are some examples of LISP forms:

```
X
(CAR X)
MYCADILLAC
(CONS MYCADILLAC (CAR X))
Y
(CDR Y)
(ATOM (CDR Y))
(CONS (ATOM (CDR Y)) (CONS MYCADILLAC (CAR X)))
```

We invite the reader to match the parentheses and check the correctness of each form against the definition.

Although some LISP systems tolerate and understand (not always the same way) "forms" such as:

```
(CONS MYCADILLAC)
(CONS X Y Z)
```

which have too few or too many arguments, we regard them as incorrect here—they are a source of trouble for LISPers, especially beginners.

No words are *reserved* in LISP except **NIL** and **T** (see Section 4.5); the same atom can be used to name a function and receive a value. Thus the atom CONS alone is an acceptable form, and so is the form (CONS CONS CONS), in which the first occurrence of CONS denotes the function and the second and third should be bound to some value.

4.3 MATHEMATICAL VERSUS LISP NOTATION

Although LISP is certainly very close to mathematics and logic, it uses a slightly different notation (which may be a sign of modern times). There is no profound significance to this difference, but it will help the reader who has some mathematical background to compare the two notations.

Where mathematicians and logicians write:

$$f(\mathrm{arg}_1, \ldots, \mathrm{arg}_n)$$

LISPers do not use commas:

$$(f\ \mathrm{arg}_1\ \ldots\ \mathrm{arg}_n)$$

In fact, mathematicians and logicians also use *infix notation*, which is available on some modern pocket calculators. The functional symbol + or * (add or multiply operator) is placed between its operands, instead of in front, as in the expressions:

```
2 + 5
3 * (2 + 5)
```

and *priority rules* may have to be invoked to avoid certain ambiguities.

Figure 4-1 shows the correspondence between the mathematical way and the LISP way.

Mathematical Way	LISP Way
MAX(X, Y)	(MAX X Y)
2 + 5	(+ 2 5)
3 * (2 + 5)	(* 3 (+ 2 5))
COS(A + B)	(COS (+ A B))
CAR(X)	(CAR X)
CDR(X)	(CDR X)
CONS(CAR(X), CDR(X))	(CONS (CAR X) (CDR X))

FIGURE 4-1. Mathematical Notation and LISP Notation

4.4 EVALUATION OF SIMPLE FORMS IN AN ENVIRONMENT

Consider an environment where the following bindings, among others, are realized:

Atom	Value
MYCADILLAC	(MY EXPENSIVE CAR)
FRIEND	PETER
ACTION	DRIVES
SPIDERMAN	*UNBOUND*

We can compute values of simple forms involving the basic functions CONS, CAR, CDR, and ATOM. When a form is an atom, the value of the form is the value of the atom. Thus the form:

```
MYCADILLAC
```

evaluates to the list:

```
(MY EXPENSIVE CAR)
```

The form:

```
(CAR MYCADILLAC)
```

evaluates to the result of applying the CAR function to the value (MY EXPENSIVE CAR) of the atom MYCADILLAC. The result is therefore the atom

```
MY
```

Similarly, the form:

```
(CDR MYCADILLAC)
```

has the value:

```
(EXPENSIVE CAR)
```

Consider the slightly more complex form:

```
(CAR (CDR MYCADILLAC))
```

What is the value of this form? We notice that actually, there is only one argument; (CDR MYCADILLAC) is a form whose value we know to be (EXPENSIVE CAR) from the previous example! To compute the final result, we need only apply the CAR function to the list (EXPENSIVE CAR). We obtain the atom:

```
EXPENSIVE
```

The form:

```
(CDR SPIDERMAN)
```

does not have any value in the above environment, since SPIDERMAN is unbound.

The form:

```
(CDR (CAR (CDR MYCADILLAC)))
```


will cause an error because the argument (CAR (CDR MYCADILLAC)) evaluates to an atom, and it is not possible to take the CDR of an atom.[2] Recall that the ATOM function returns T (which means "true") when applied to an atom and NIL (which means "false") when applied to a nonatomic object: Thus, the form:

```
(ATOM MYCADILLAC)
```

has the value:

```
NIL
```

because the value of MYCADILLAC is a dotted pair, not an atom.

Here is a form involving the CONS function:

```
(CONS ACTION MYCADILLAC)
```

The value of this form is the result of CONSing the value DRIVES of the first argument ACTION to the value (MY EXPENSIVE CAR) of the second argument MYCADILLAC, that is:

```
(DRIVES MY EXPENSIVE CAR)
```

Using the more complex form:

```
(CONS MYFRIEND (CONS ACTION MYCADILLAC))
```

we produce the English sentence:

```
(PETER DRIVES MY EXPENSIVE CAR)
```

4.5 NIL, T, AND NUMBERS EVALUATE TO THEMSELVES

The value of NIL is NIL in all environments.

The value of T is T in all environments.

NIL and T are thus predefined. Any attempt to change their values will give an error message.

Numbers are also self-evaluating–for instance, the value of 1982 is 1982. (In some LISP systems, strings are self-evaluating: thus, the string "Hi there" is considered the same as the literal atom HI THERE (but you need the " " so that the system sees one atom instead of two); the value of this atom is automatically set to the atom HI THERE itself.)

Figure 4-2 shows an environment and additional forms and their evaluation: the reader may want to perform these evaluations and check the results.

4.6 THE VALUE OF A FORM CAN BE RECURSIVELY DEFINED

It should now be clear to the reader that the evaluation of a form goes from *inside* to *outside*. For simple forms such as the ones given in Section 4.4, we can give the following semi-formal definition.

Definition 2: The *value* of a form *f* in an environment *e* is

the value of the atom *f* in the environment *e*, if *f* is atomic

---

[2] Some LISP implementations do let you take the CAR or CDR of an atom, but usage of this possibility is bad programming practice.

Atom	Value
MINE	(TWIN FIN SURFBOARD)
YOURS	(SINGLE FIN SURFBOARD)
FIN	***UNBOUND***
PARTY	(WINE AND CHEESE)
CHEESE	(AGED SWISS)
WINE	(CALIFORNIA CHABLIS)

Form	Value
(CDR (CDR PARTY))	(CHEESE)
(CAR (CDR (CDR PARTY)))	CHEESE
(CDR (CDR (CDR PARTY)))	NIL
(CAR PARTY)	WINE
(CAR (CAR PARTY))	***ERROR*** Attempt to take the CAR of the atom WINE
(CAR WINE)	CALIFORNIA
(CONS (CAR WINE) (CAR PARTY))	(CALIFORNIA . WINE)

Form	Value
(CONS (CAR WINE) PARTY)	(CALIFORNIA WINE AND CHEESE)
(CONS PARTY CHEESE)	((WINE AND CHEESE) AGED SWISS)
(CAR MINE)	TWIN
(CDR YOURS)	(FIN SURFBOARD)
(CONS (CAR MINE) (CDR YOURS))	(TWIN FIN SURFBOARD)
(CONS (CDR YOURS) (CAR MINE))	((FIN SURFBOARD) . TWIN)
(ATOM MINE)	NIL
(ATOM (ATOM MINE))	T

FIGURE 4-2. Evaluation of Simple Forms in a Given Environment

the result of applying the function s to the values of the arguments $arg_1, \ldots, arg_n$ in the environment e if f is the form $(s\ arg_1 \ldots arg_n)$, where s is the name of a function and all the arguments have values

undefined otherwise

4.7 QUOTE AND ' OR HOW TO PREVENT EVALUATION

In order for a form to evaluate properly, all the atoms occurring in argument positions should have values. In the form:

```
(F X (G Y Z))
```

the atoms X, Y, and Z should have values because they are arguments. On the other hand, F and G need not have values, because they are not arguments; they should designate existing functions of arity 2.

The **QUOTE** function is a very special function in LISP, which does not follow our earlier description of the evaluation process. QUOTE is thus an exception, but an extremely useful exception. QUOTE takes one argument, and *this argument is not evaluated!*

The value of (QUOTE x) is simply x itself (not the value of x), whatever S-expression x is.

QUOTE is thus a means of preventing evaluation. If we desire to refer to the atom HELLO itself, not to the value of HELLO, we shall write (QUOTE HELLO) instead of HELLO. Suppose that we want to say HELLO to some friend, and we are in an environment where MYFRIEND is bound to (MARY LOU). We type:

```
(CONS (QUOTE HELLO) MYFRIEND)
```

and the value returned by LISP will be:

```
(HELLO MARY LOU)
```

For the same result, we could have typed:

```
(CONS (QUOTE HELLO) (QUOTE (MARY LOU)))
```

or simply:

```
(QUOTE (HELLO MARY LOU))
```

The first formulation is preferable because of its generality: it would work for different friends, i.e., different values of MYFRIEND. Note that typing:

```
(CONS HELLO MYFRIEND)
```

would be a mistake if HELLO was unbound. Typing:

```
(CONS (QUOTE HELLO) (QUOTE MYFRIEND))
```

would not be a mistake but would prevent the evaluation of MYFRIEND, and thus yield the (probably unwanted) result:

```
(HELLO . MYFRIEND)
```

The reader may ask the question, Is QUOTE really necessary? If we had an atom H bound to HELLO, in addition to MYFRIEND being bound to (MARY LOU), then we could type, with no QUOTE:

```
(CONS H MYFRIEND)
```

instead of:

```
(CONS (QUOTE HELLO) MYFRIEND)
```

Why not do this? There is a good reason:

Environments and bindings do not emerge from nothing!

So far, we have not seen how to set up environments, and bindings in particular. Let us just say that binding H to HELLO would necessitate the use of QUOTE or some other function based on the same idea. In other words, we could avoid QUOTE here only at the expense of using it somewhere else.

Taking the other extreme view, the reader may decide that QUOTE is good enough to supersede all other functions. Indeed, one can obtain any wanted result *r* by typing

```
(QUOTE r)
```

For example:

```
(QUOTE (THIS IS THE (EXACT) RESULT I WANT))
```

will produce:

```
(THIS IS THE (EXACT) RESULT I WANT)
```

Of course, this does not lead us very far toward useful programming. In LISP, as in other programming languages, a form or a program is useful only if it can produce various results that depend on the environment.

A **QUOTE-form**, that is, a form starting with QUOTE, has a unique value which does not depend on the environment. *No evaluation, at any level whatsoever, is performed inside a QUOTE-form*. For these reasons, QUOTE-forms are often called *constants*.

In many programming languages, the main, if not the only, constants are numbers. There is no need to QUOTE, because there is an obvious difference between identifiers (the equivalent of LISP literal atoms) and numbers. When we write SIZE + 3 in FORTRAN, we really mean "the result of adding 3 to *the value of* SIZE." The variable SIZE implicitly refers to the value of SIZE, whereas the constant 3 stands for itself (3 cannot have the value 5!).

In LISP there is a need for explicit QUOTing, because constants could look like variables otherwise, or even like forms in general. However, numbers do not need to be QUOTEd in LISP: the value of 17 is 17, and so is the value of (QUOTE 17). Strings, when they exist, may or may not need to be QUOTEd, depending on the LISP system.

Since *the value of NIL is NIL, it is not useful to quote NIL*. (The value of (QUOTE NIL) is also NIL.)

Since *the value of T is T, it is not useful to quote T.*

A feature of most LISPs is the **'-notation.** It greatly alleviates the burden of QUOTing.

'x is an abbreviation for (QUOTE *x*), whatever S-expression *x* is.

The sign ' is certainly shorter to type than QUOTE. The '-notation also saves a pair of parentheses. Note that no space should be left between ' and the quoted S-expression. For example:

```
'HELLO
```

is equivalent to:

```
(QUOTE HELLO)
```


and:

```
(CONS 'HELLO MYFRIEND)
```

is equivalent to:

```
(CONS (QUOTE HELLO) MYFRIEND)
```

The symbol ' may be confusing for beginners, because it is an exception to the universal LISP syntax. It is also less visible than QUOTE. The LISP system translates '-notation into QUOTE-notation before any evaluation is performed, which may cause some surprises at first. For instance, the value of:

```
'(CONS 'X (CDR Y))
```

will be understood as:

```
(QUOTE (CONS (QUOTE X) (CDR Y)))
```

and will be printed in QUOTE-notation as:

```
(CONS (QUOTE X) (CDR Y))
```

rather than:

```
(CONS 'X (CDR Y))
```

Because LISP prints values in list notation whenever possible, the value printed for (QUOTE *x*) may look different from *x*, but they both represent the same object. For example:

```
(QUOTE (A . (B . (C . NIL))))
```

will have its value printed as:

```
(A B C)
```

The QUOTE function may be used to obtain the list form of proper lists typed in dot notation or to check whether a given dotted pair is a proper list. (If no periods remain in the printed value, then the dotted pair is a proper list.)

4.8 SETTING UP AN ENVIRONMENT WITH SETQ

All the primitives introduced so far—ATOM, CAR, CDR, CONS, and QUOTE—leave the environment unchanged. The **SETQ** function provides a means of altering or setting up environments.

SETQ takes two arguments. The first argument will not be evaluated; it should be an atom. If *x* is an atom and *y* is a form, then:

- The value of (**SETQ** *x y*) in an environment *e* is the value of the second argument *y* in the same environment *e*.
- There is an important side-effect: the environment *e* is changed in the following manner. The atom *x* gets bound to the value of *y* (whether or not *x* was already bound). The atom *x* must be different from NIL and T, which cannot be reset.

For example, to set up the environment at the beginning of Section 4.4, we could type the three *SETQ-forms:*

```
(SETQ MYCADILLAC '(MY EXPENSIVE CAR))
(SETQ FRIEND 'PETER)
(SETQ ACTION 'DRIVES)
```

The particularity of SETQ not to evaluate its first argument corresponds to the most commonly used form of binding: except on rare occasions, one wants to bind a literal atom rather than bind its value. If the latter is desired, the **SET** function is the one to be used. In fact, SETQ is only a luxury since the effect of the form:

```
(SETQ x y)
```

can be achieved using:

```
(SET 'x y)
```

4.9 SUPERBRACKETS LESSEN THE RISK OF PARENTHESES MISTAKES

Most LISP systems provide *superbrackets*: "[" and "]" (or "<" and ">", depending on the dialect). A pair of superbrackets can replace a pair of matched parentheses anywhere. Superbrackets and parentheses can be nested at will. The fundamental and very useful property of superbrackets is that:

The closing superbracket "]" implicitly closes parentheses that were opened after the corresponding opening "[" but not closed.

For example:

```
(HELLO (MY (DEAR FRIEND)))
```

could be typed:

```
[HELLO (MY (DEAR FRIEND]
```

The list:

```
(HELLO (MY (DEAR FRIEND)) BYE)
```

could be typed:

```
(HELLO [MY (DEAR FRIEND)] BYE)
```

or:

```
[HELLO [MY (DEAR FRIEND] BYE]
```

However, superbrackets are nothing more than a convenient typing feature. Other competing modern features are the automatic *pretty formatting* of your input (showing the structure of what you are typing by means of an appropriate indentation) and *automatic flashing* of "(" when you close it with ")". Program editors that know about LISP should be used whenever extensive typing is done in LISP. The lack of such features in some LISP systems may be sufficient to discourage the most enthusiastic students!

4.10 PRETTYPRINTING IS A MUST

Complex forms are illegible unless they are presented in a way that shows their structure. A prettyprinting function is available in good systems. Obviously:

```
(CONS (CDR (MEMBER U V))
      (CONS (CAR (CDR X))
            (APPEND Y Z)))
```

is much clearer than:

```
(CONS (CDR (MEMBER U V)) (CONS (CAR (CDR X)) (APPEND Y Z)))
```

which is nevertheless the same form.

5. PROGRAMMING IN LISP

As we saw in Chapter 4, it is possible to converse with a LISP system without ever writing a program. These conversations can be very instructive; they can check the user's knowledge of basic LISP functions or be used to experiment with function packages offered with LISP systems.

To go beyond this point–for example, to solve some substantial problem–it is necessary to write programs, or functions. We prefer the term of *function* because it is closer to the "spirit of LISP." Once a function has been satisfactorily defined, it can be used to define other functions. As we hinted in Chapter 2, a function may eventually be defined in terms of itself–a *recursive function*.

In this chapter we present the additional ingredients necessary to write and use LISP functions in keeping with the spirit of LISP.

5.1 LISP IS A FUNCTIONAL LANGUAGE

Mathematicians are familiar with the concept of function and the idea of *composing* functions to build more complex ones. This powerful idea has somehow been kept out of sight in the earliest programming languages. Although FORTRAN does not condemn this method, the nature of the language does not encourage it: a FORTRAN program generally looks like a sequence of instructions to be executed step by step. The role of each instruction is to alter the value of some variable; at the end, a key variable will typically hold the desired result. This is called the *Von Neumann approach to computing*.

An *applicative language* such as LISP handles a program as a mathematical function. It produces a result by applying some treatment to the input. The treatment may consist of just one basic function, or it may consist of several layers of transformations (each one applied to the input, or on top of some other one).

To illustrate our point, we shall use the example of a recipe for Italian scallops. Recipes are usually given in a Von Neumann style, which we shall use at first:

1. Preheat oven at 350° F.
2. Boil eggplant for 2 minutes.
3. Chop scallops in food processor.
4. Mix with eggplant.
5. Bake 30 minutes in oven at 350° F.

In a LISP style, the same recipe is considered as a transformation of an eggplant and scallops into the delicious Italian dish. It could look like:

```
(BAKE (MIX (BOIL 'EGGPLANT 2)
           (CHOP 'SCALLOPS))
      30
      (PREHEAT 'OVEN 350) )
```

where BAKE, MIX, BOIL, and CHOP are basic cooking operations–just as CONS, CAR, CDR, and ATOM are basic LISP functions–intuitively defined as follows:

(BAKE *f t c***)**	result of baking food *f* for *t* minutes in cooking device *c*
(MIX *f*1 *f*2**)**	mixture of food *f*1 and food *f*2
(BOIL *f t***)**	food obtained by boiling food *f* for *t* minutes
(CHOP *f***)**	food *f* after it has been hashed
(PREHEAT *c h***)**	cooking utensil *c*, with temperature raised to *h*° F.

The first form of the recipe is probably preferred by most cooks, but it hides the logical structure of the operations–for example, it arbitrarily requires boiling the eggplant (step 2) before chopping the scallops. Clearly, these two steps are independent and could be performed in any order, or simultaneously.

The second form of the recipe unveils its structure: here, (BOIL 'EGGPLANT 2) and (CHOP SCALLOPS) are arguments of the MIX function. They are placed at the same level: no operational ordering is arbitrarily imposed. CHOP and BOIL should take place before MIX, and MIX and PREHEAT before BAKE. This is simple common sense. The functional diagram in Figure 5-1 illustrates this dependency.

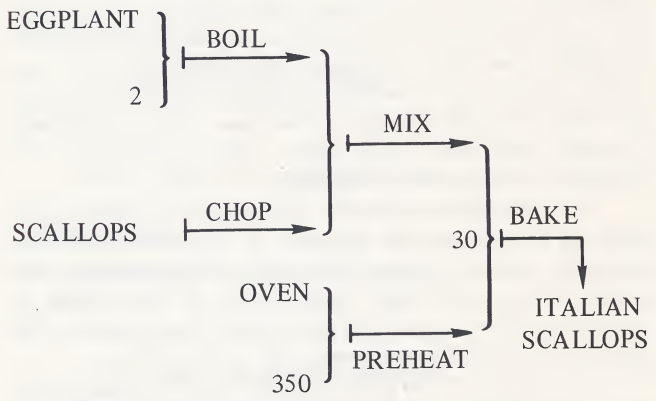


FIGURE 5-1. Functional Diagram For a Scallop Recipe

Another difference between these two styles deserves some attention. Each step in the first form of the recipe carries a partial transformation of some ingredient. For example, the eggplant in Step 4 is not really the same as in Step 2. This fact is implicit in English; in a real programming language, intermediate results would have to be explicitly saved in some identified location, i.e., EGGPLANT would have to be treated as a variable. The functional approach offered by LISP makes this tedious bookkeeping automatic and totally transparent to the programmer.

5.2 TRUTH AND FALSITY IN LISP

For logical purposes, the atom NIL means "false." "True" is represented by anything other than NIL–for instance, the atom T. (By convention, the atom T is always bound to T, just as NIL is always bound to NIL. Any attempt to change the value of NIL or T will give an error message.) *T is only one in a million representatives of the truth!*

This feature turns out to be very convenient in practice. Instead of forcing predicates to return either NIL or T, we can let them return any nonNIL value when they should return T. This nonNIL value can be used to convey additional information besides "truth."

A typical example is the **MEMBER** predicate. MEMBER tells you whether a given S-expression belongs to a given list: for instance, B belongs to the list (A B C), while D does not. Instead of returning T as an answer to MEMBER, we can return (B C), which means "true" since it is not NIL. In general, let MEMBER return *the first non-empty sublist starting with the given S-expression, or NIL if it does not belong*. This implementation turns out to be simpler than the less informative one.

5.3 TESTS USING THE COND-FORM

Any programming language provides a means of testing a condition. Depending on the result of the test, different actions will be performed. **COND** answers this need in LISP. The syntax of COND is a little bizarre (for historical reasons) and is another exception to the general case:

- COND does not have a fixed number of arguments.
- Each argument of COND is not a form but a pair of (i.e., a list of two) forms.

In general, a *COND-form* looks like:

```
(COND (test1 result1)
      ...
      (testn resultn))
```

where n is the number of pairs and $test_1, \ldots, test_n$, $result_1, \ldots, result_n$ are LISP forms.

The value of this COND-form is the value of the first $result_i$, $1 \leq i \leq n$, for which the corresponding $test_i$ has a nonNIL value. Note that it is not required that the value of the test be T (true)!

Should it happen that all the tests evaluate to NIL, the value of the COND-form is NIL. However, this seldom happens in practice, as it is customary that the last test be T, so that at least one test will succeed.

Let us consider some examples in the following environment:

Atom	Value
MARGARITA	(SPIRITS (LEMON JUICE))
SPIRITS	(TEQUILA TRIPLESEC)
STEVEN	DRUNK
FRUITS	NIL

The COND-form

```
(COND (STEVEN (CAR (CDR MARGARITA)))
      (T      (CAR MARGARITA)))
```

contains two pairs and means: If the value of STEVEN is not NIL, then return the CAR of the CDR of the value of MARGARITA (that is, (LEMON JUICE)); otherwise return the CAR of the value of MARGARITA (that is, SPIRITS).

Since the value of the first test STEVEN is DRUNK, the value of the above COND-form is (LEMON JUICE).

Consider the COND-form:

```
(COND (FRUITS          (CAR SPIRITS))
      ((ATOM STEVEN)   MARGARITA)
      (T               'BANANA))
```

which has three pairs. It reads: If the value of FRUITS is not NIL, then return the CAR of the value of SPIRITS (that is, TEQUILA); if the value of STEVEN is atomic, return the value of MARGARITA; otherwise return the atom BANANA.

Clearly, the first test FRUITS will fail, since its value is NIL. The second test (ATOM STEVEN) evaluates to T (which is not NIL). Therefore the value of the COND-form is that of MARGARITA: (SPIRITS (LEMON JUICE)).

It should be noted that a COND-form is also a LISP form. Therefore it can occur wherever a LISP form is expected, which is almost everywhere. For instance, a test in a COND-form could be a COND-form; an argument of a function could be a COND-form, as in:

```
(CONS STEVEN
      (COND (FRUITS SPIRITS)
            (T      'PEAR)))
```

which is equivalent to the less elegant:

```
(COND (FRUITS (CONS STEVEN SPIRITS))
      (T      (CONS STEVEN 'PEARS)))
```

This is a very important characteristic of LISP, shared by few other programming languages. COND should therefore be treated like any other LISP function, except for its bizarre syntax.[1]

5.4 NULL, AND, AND OR ARE USEFUL PREDICATES

These functions can be defined in terms of COND. They can simplify LISP coding.

NULL is a function of one argument. (NULL x) is equivalent to the form:

```
(COND (x NIL) (T T))
```

The value of (NULL x) is thus T when x's value is NIL, and NIL when x's value is not NIL. NULL may be useful in some situations, but it is often misused.

The form:

```
(COND ((NULL x) y) (T z))
```

is equivalent to:

```
(COND (x z) (T y))
```

but is, obviously, more complex![2]

AND and **OR** again are special: they can take any number of arguments.

- (AND x_1 ... x_n) evaluates to "true" if all the arguments $x_1, \ldots, x_n$ have a nonNIL value, and NIL if any one of the arguments evaluates to NIL. But instead of "true" being represented by T, it is represented by the (nonNIL) value of the last argument, x_n! This is logically equivalent but has the merit of eventually carrying more information, although not all LISP systems do it this way. (When an argument evalutes to NIL, the arguments that follow are not evaluated.)

- (OR $x_1, \ldots, x_n$) evaluates to the value of the first argument from x_1 to x_n which does not evaluate to NIL, if there is one (in which case evaluation of the arguments stops), and to NIL otherwise.

For example, the form:

```
(AND MARGARITA STEVEN)
```

evaluates to DRUNK in the environment given in Section 5.3, since DRUNK is the value of STEVEN and MARGARITA has a nonNIL value.

The form:

```
(OR FRUITS MARGARITA STEVEN)
```

evaluates to:

```
(SPIRITS (LEMON JUICE))
```

which is the value of MARGARITA (STEVEN is not evaluated).

```
(AND FRUITS MARGARITA STEVEN)
```

would evaluate to NIL since this is the value of FRUITS. (MARGARITA and STEVEN would not be evaluated).

Clever usage of AND, OR, and NULL may simplify programs by eliminating several CONDs (see, for example, the definition of EQUAL given in Section 5.6). Note that the nonevaluation of some arguments is significant if their evaluation could change the environment (e.g., use of a SETQ-form in an argument). The order of their evaluation, from left to right, is often significant even in the absence of side-effects.

5.5 EQ IS A DANGEROUS LISP PRIMITIVE

EQ, like ATOM, is a basic LISP predicate. It takes two arguments.

The value of the form (EQ x y) is T if the value of the form x is represented at the same address in the computer memory as the value of the form y; it is NIL otherwise.

EQ is thus very close to the machine implementation of LISP. Curiously, the EQ function is not very reliable! In all LISP systems, literal atoms have a unique representation, their address in the symbol table. Thus, if we know that x and y have the same literal atom as their value, we can be sure that (EQ x y) will evaluate to T. For instance:

```
(EQ 'HELLO 'HELLO)
```

evaluates to T. But nothing is sure for numbers: most LISP systems represent only small numbers in a unique way, but the notion of *small* obviously depends on the implementation. The form:

```
(EQ 18715 18715)
```

would evaluate to NIL if 18715 was considered a big number and to T if it was a small number.

In the environment of Section 5.3, the form:

```
(EQ MARGARITA MARGARITA)
```

---

[1] One other difference will be studied in Section 5.11.

[2] Some LISPers prefer the more complex form in order to get rid of the null case first and then be able to concentrate on the more difficult one.

evaluates to T because the address of the representation of the value of MARGARITA is obtained by consulting the MARGARITA entry in the symbol table. Since this entry is unique, this address has no reason to be different the second time. On the contrary, typing:

```
(EQ '(TEQUILA (LEMON JUICE))
    '(TEQUILA (LEMON JUICE)))
```

will yield a surprising NIL. The reason is not LISP alcoholism! Simply, the typed form is represented in memory before evaluation: the first sublist (TEQUILA (LEMON JUICE)) is at one address, while the second one is at a different address.

5.6 EQUAL IS A SAFE MEANS OF TESTING EQUALITY

EQ is interesting in theory because **EQUAL** can be (recursively) defined in terms of EQ and the other primitives CAR, CDR, ATOM, and COND, at least in pure LISP, where the question of small versus big numbers is irrelevant.

EQUAL should be used instead of EQ for most applications.

EQUAL corresponds to the mathematical notion of equality: *two objects are equal if they look alike*. For example, the list (TEQUILA (LEMON JUICE)) is clearly equal to the list (TEQUILA (LEMON JUICE)). More precisely, two LISP objects are EQUAL if they have the *same structure* (and would, therefore, print the same). Hence, the S-expression:

```
(TEQUILA . ((LEMON . (JUICE . NIL)) . NIL))
```

is EQUAL to the previous list.

The syntax of EQUAL is the same as that of EQ. The value of (EQUAL *x y*) is T if the forms *x* and *y* evaluate to equal S-expressions (i.e., expressions that have the same structure); otherwise, they evaluate to NIL.

As an illustration of how most LISP functions can be defined in terms of a few LISP primitives, let us define EQUAL in terms of EQ, CAR, CDR, LISTP, and COND. (We use AND and OR for the sake of clarity, but they could be replaced with CONDs.) Assuming we are dealing with pure LISP, the definition could be:

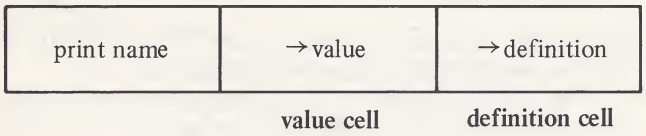
```
(DEFINE 'EQUAL '(X Y)
        '(OR (EQ X Y)
             (AND (LISTP X)
                  (LISTP Y)
                  (EQUAL (CAR X) (CAR Y))
                  (EQUAL (CDR X) (CDR Y)))))
```

which reads: X is **EQUAL** to Y *if and only if* X is EQ to Y or X and Y are both dotted pairs and the CAR of X is (recursively) **EQUAL** to the CAR of Y, and the CDR of X is (recursively) **EQUAL** to the CDR of Y.

5.7 DEFINING FUNCTIONS WITH DEFINE

We have now seen several functions–CAR, CDR, CONS, ATOM, COND, NULL, AND, OR, EQUAL. Good LISP systems offer hundreds, even thousands, of such *predefined* functions, which can be combined into simple or complex forms. Obviously, the user will want to define his or her own functions. Fortunately, this is possible in LISP!

We have seen in Section 4.2 that literal atoms are stored in the *table of atoms* and may have a value. This value is held by the so-called *value cell* of the atom. Similarly, a *function definition* may be attached to an atom through its *definition cell*. The value cell and definition cell are, in most LISPs, two distinct fields of the atom entry in the symbol table:



The value cell and the definition cell actually contain pointers to (i.e., memory addresses of) the value or definition of the atom. Just as SET or SETQ allows us to attach a value to an atom, the **DEFINE** function (or some similar function) can be used to attach a function definition to an atom. More precisely, evaluation of the form:

```
(DEFINE fname paramlist body)
```

changes the environment by attaching the values of *paramlist* and *body* to the atomic value of *fname*. Thus

fname should evaluate to a literal atom–the name of the function to define (or redefine)

paramlist should evaluate to a list of distinct literal atoms (each one different from NIL and T)–the parameters

body should evaluate to a LISP form–the function body; this body normally contains occurrences of all the parameters

When everything is correct, the value of the *DEFINE-form* is typically a list of the name of the function that has been defined or redefined.

In the next section we introduce algebraic expressions: this context, for which LISP is especially well suited, illustrates use of function definitions and function calls (see Section 5.9).

5.8 A FEW FUNCTIONS FOR DEALING WITH SYMBOLIC EXPRESSIONS

Algebraic expressions such as

```
(X + Y)
(A * (B + C))
((COS A) – (SIN B))
```

are easy to represent in LISP by means of lists. Thus, the list (A * (B + C)) is made up of three components:

Left operand: A, an atomic expression

Main operator: *

Right operand: (B + C), composed of:

Left operand: B, an atomic expression

Operator: +

Right operand: C, an atomic expression

For simplicity, we do not consider expressions that are not fully parenthesized, e.g.:

```
(A + B + C),
(A + B * C),
```

which we will write:

```
((A + B) + C),
(A + (B * C)),
```

respectively, according to usual mathematical conventions.

We write (COS A) rather than COS(A).

We can now specify three LISP functions for selecting the components of any nonatomic expression:

(LEFT EXP) returns the operand to the left of the main operator of the expression (which is the value of EXP); NIL if there is no left operand:

(LEFT '(A * (B + C))) evaluates to A

(LEFT '(COS A)) evaluates to NIL

(OP EXP) returns the main operator of the expression:

(OP '(A * (B + C))) yields *

(OP '(COS A)) yields COS

(RIGHT EXP) returns the operand to the right of the main operator:

(RIGHT '(A * (B + C))) evaluates to (B + C)

(RIGHT '(COS A)) evaluates to A

To define these functions in LISP is quite easy, if we note that there are basically two cases:

Case 1. The expression is a list of two elements, e.g., (COS A). The first element–the main operator–is obtained by taking the CAR of the expression; the second element–the right operand–is the CAR of the CDR of the expression.

Case 2. The expression is a list of three elements, e.g., (A * (B + C)). The first element–the left operand–is the CAR of the expression; the second one–the main operator–is the CAR of the CDR of the expression; the third one–the right operand–is obtained by taking the CAR of the CDR of the expression's CDR.

To distinguish between these two cases, it suffices to test whether the CDR of the expression is NIL (Case 1) or not (Case 2). Hence the following definitions:

```
                                        Comments
(DEFINE 'LEFT                           function name
        '(EXP)                          list of parameters
        '(COND                          body starts here
              [(CDR (CDR EXP))           [test
               (CAR EXP)]                 result]        (case 2)
              [T   NIL]))                [test result]  (case 1)

(DEFINE 'OP                             function name
        '(EXP)                          list of parameters
        '(COND                          body starts here
              [(CDR (CDR EXP))           [test
               (CAR (CDR EXP))]           result]        (case 2)
              [T (CAR EXP)]))            [test result]  (case 1)
```



```
(DEFINE 'RIGHT                          function name
        '(EXP)                          list of parameters
        '(COND                          body starts here
              [(CDR (CDR EXP))           [test
               (CAR (CDR (CDR EXP)))]     result]        (case 2)
              [T (CAR (CDR EXP))]))      [test result]  (case 1)
```

5.9 USING FUNCTIONS THAT HAVE BEEN DEFINED

From the user's point of view there is no essential difference between newly created functions and predefined functions. Suppose one has defined the function OP as in Section 5.8 and wants to apply OP to some expression, e.g., (A * (B + C)). One may type:

```
(OP '(A * (B + C))).
```

Such a form will be evaluated according to the following fundamental definition:

Definition 1. Let f be a function of n parameters $(x_1, \ldots, x_n)$ and body b. Let $\text{arg}_1, \ldots, \text{arg}_n$ be n LISP forms. Then the value of the form (we say, *function call*):

$$(f\ \text{arg}_1 \ldots \text{arg}_n)$$

in environment e is the value of b in environment e, except for the bindings of x_1 to arg_1's value in e, x_2 to arg_2's value in e, and so on up to x_n bound to arg_n's value in e; $\text{arg}_1, \ldots, \text{arg}_n$ are called the *arguments* of f in the function call and yield * as a result of the evaluation.

A function may have zero, one, two, or more parameters. It should always be given the corresponding number of arguments when used.[3] *Supplying fewer or more arguments may have different effects depending on the LISP system.* It is not recommended.

This definition is fundamental and independent from any particular implementation. However, it is very instructive to understand how real LISP interpreters evaluate function calls. This is the subject of the next two sections.

5.10 LAMBDA-EXPRESSIONS ARE THE INTERNAL REPRESENTATION OF DEFINITIONS

LISP stores definitions under a special form called *LAMBDA-expressions,* from the theory of Lambda Calculus. Although LISP's relation to this theory is rather accidental, it is necessary to know about LAMBDA-expressions because this is what we see when we have access to the definition (through functions such as GETD or GETFN, which return the definition field of an atom, or through LISP editors). In some dialects, DEFINE or its equivalent expects a LAMBDA-expression as one of the arguments.

Let f be the name of a function whose parameter list is $(p_1, \ldots, p_n)$ and whose body is b. The list:

$$(\text{LAMBDA}\ (p_1, \ldots, p_n)\ b)$$

[3] Most LISP systems also support a type of function called SPREAD, where the number of arguments is variable.

called a LAMBDA-expression (or *function definition*), will be attached to the atom *f* when the function is defined.

An important property of the LAMBDA-expression attached to a function name is that it is strictly equivalent to the name! For instance, the form:

```
([LAMBDA (EXP)
         (COND [(CDR (CDR EXP))
                (CAR (CDR EXP))]
               [T    (CAR EXP)])]
 '(A * (B + C)))
```

is equivalent to:

```
(OP '(A * (B + C)))
```

provided that OP is defined as in Section 5.8.

By means of LAMBDA-expressions one can use a function without having named it through a DEFINE-form. The utility of this feature will be seen in Section 6.10.

5.11 HOW LISP EVALUATES FUNCTION CALLS

To evaluate a function call, LISP systems use different methods. One of the most popular is based on a temporary binding (called *shallow binding*) of the parameters to the values of the arguments. The environment is modified during the evaluation but is restored at the end.[4] We consider a function named *f* with parameters $p_1, \ldots, p_n$ and body *b* (*f* may be replaced with the corresponding LAMBDA-expression in the statement below.)

To **evaluate** (f $\arg_1, \ldots, \arg_n$), take the following steps:

1. Save the old values of the parameters (that is, the values of $p_1, \ldots, p_n$ currently available from the symbol table.
2. **Evaluate** arguments $\arg_1, \ldots, \arg_n$.
3. Bind each parameter to the value of the corresponding argument (that is, p_1 to $\arg_1$'s value, ..., p_n to $\arg_n$'s value). The environment (the symbol table) is now changed.
4. **Evaluate** body *b* of function *f* in this new environment.
5. Restore the old values of the parameters (that is, rebind the parameters to the values saved in Step 1). The environment is now restored.
6. Return the value computed in Step 4 as the final value of the function call.

The above process is deeply recursive, because the arguments (evaluated in Step 2) and the body of *f* (evaluated in Step 4) may also be function calls. However, if no recursive function is encountered, the process is ensured to stop because it eventually reaches atomic forms or forms involving only primitive functions.

[4] Only the parameters are restored; modifications introduced by calls to DEFINE, for example, are permanent.

As an illustration, let us consider function RIGHT defined in Section 5.8. To make things clearer, we shall redefine it exclusively in terms of user-defined functions: as shown below, MYCAR and MYCDR are just synonyms of CAR and CDR. We also simplify RIGHT a little by considering only nonatomic expressions with two operands around the main operator: this allows us to remove the COND.

```
(DEFINE 'RIGHT
        '(X)
        '(MYCAR (MYCDR (MYCDR X))))
(DEFINE 'MYCAR '(X) '(CAR X))
(DEFINE 'MYCDR '(Y) '(CDR Y))
```

For the sake of variety, we have named Y the parameter of MYCDR. This does not imply any different meaning, provided that Y is used accordingly in the body, as is the case here. For a more interesting course of events, we have replaced EXP with X in the definition of RIGHT.

Figure 5-2 shows the process of evaluation of the function call (RIGHT '(A * (B + C))). It is assumed that X and Y are initially unbound. Circled numbers, linked two by two, indicate the level of nesting for each "save and restore" pair, e.g., 3 and 3'.

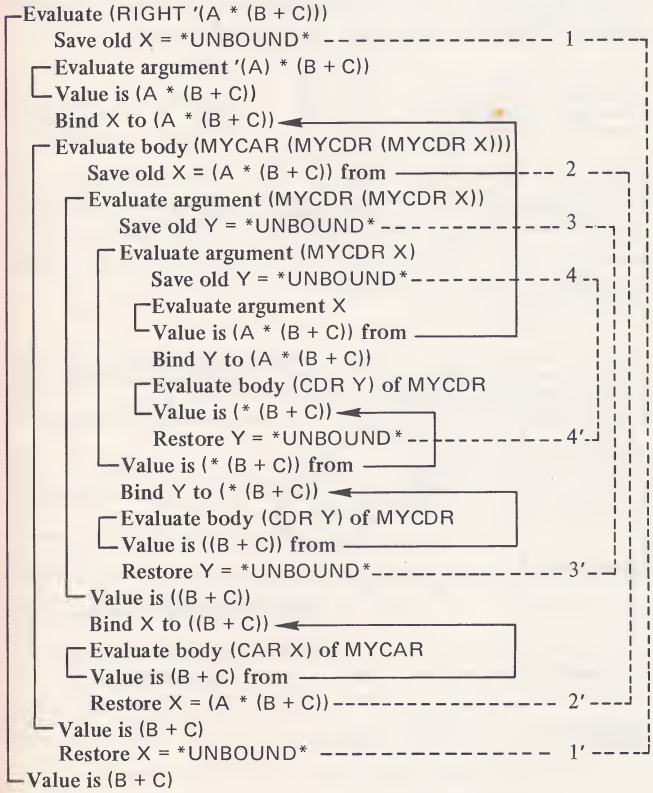


FIGURE 5-2. Evaluation of a Function Call

The reader will note up to five nested evaluations for this very simple example! A computer is keenly desired. Most LISP systems provide a *trace* function, which outputs the evaluation process in various forms. Typing something like (TRACE '(RIGHT MYCAR MYCDR)) before (RIGHT '(A * (B + C))) would have such an effect. Traces are very helpful for debugging functions and understanding the evaluation process.

Serious LISPS, e.g., INTERLISP or the MIT LISP-machine LISP, provide a *break package* that allows the user to place conditional (or unconditional) *breakpoints* in created functions. The evaluation can thus be interrupted at a critical point, and the user can inquire about the environment by typing appropriate forms or even "fix" the environment before proceeding with the evaluation. Not only the symbol table is accessible, but also the stack of saved values and nested calls.

Such powerful features are typically nonexistent in classical programming languages that require a compiler, because they are much more difficult to implement.

5.12 A STACK IS THE HEART OF LISP INTERPRETERS

Efficient LISP interpreters use a *stack*. A good illustration of a stack is the plate distributor found in some coffee shops: one can only remove plates from the top of the stack or add plates on top of the stack. Thus, the *l*ast plate *i*n is the *f*irst *o*ut: hence the term *LIFO* structures for stacks.

Steps 1 and 5 of the evaluation process described in Section 5.11 save and restore, respectively, the old values of the parameters. Saving the old values corresponds to a plate addition–we say *PUSH* in computer jargon; restoring them is similar to a plate removal–we call it a *POP*.

Figure 5-3 shows the flow of information between the symbol table and the stack. While a given literal atom *p* never has more than one entry in the symbol table, it may have several entries in the stack.

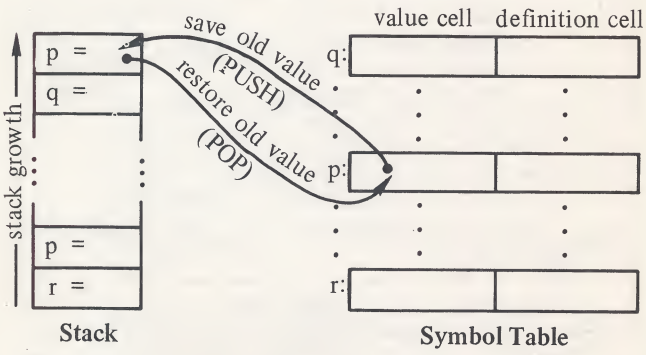


FIGURE 5-3. Save and Restore

It is important to understand that values stored in the stack are never used by the interpreter except during the restore operations; otherwise, whenever the evaluation of an atomic form is required, the value is fetched from the unique entry of the atom in the symbol table.

In Figure 5-4 we can follow the evolution of the stack and symbol table during the evaluation of the form (RIGHT '(A * (B + C))) after each of the saves and each of the corresponding restores shown in Figure 5-2. Since only X and Y are concerned, we show no other entry in the symbol table.

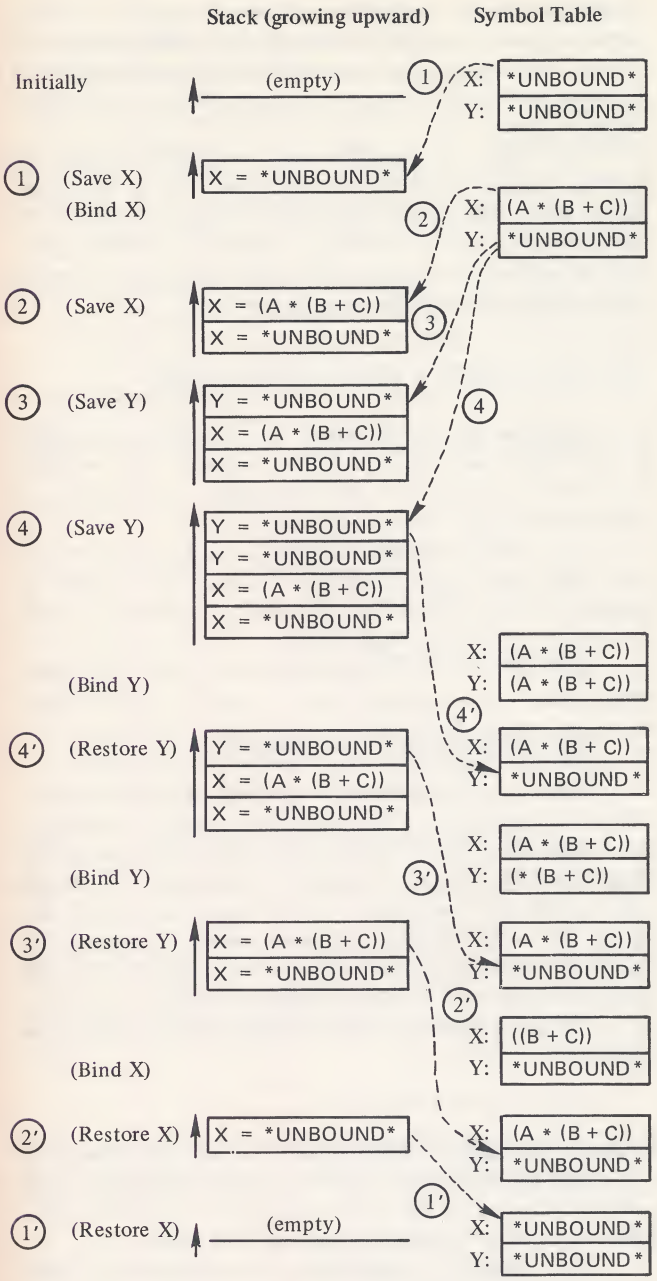


FIGURE 5-4. Evolution of the Stack and Symbol Table During the Evaluation Shown in Figure 5-2

Functions with more than one parameter do not cause any particular problem. With respect to the stack, two equivalent and quite similar methods are conceivable:

1. Create one "big entry" for storing the old values of all the parameters; then restore all the parameters from this entry.

2. Create one entry per parameter, pushing them in the order of the parameters; then restore each parameter in reversed order (since the top of the stack corresponds to the last parameter).

In either case, it would be a mistake to think that the binding of a parameter could occur immediately after its old value has been saved: none of these bindings (specified

in Step 3 of the evaluation process in Section 5.11) is performed until all the arguments have been evaluated (Step 2); otherwise, the arguments would not be evaluated in the same initial environment, as specified in Definition 1 (Section 5.9).

However, the situation may be more intricate–at least in impure LISP–if the evaluation of an argument causes some side-effect such as the binding of one of the parameters; then the remaining arguments will be evaluated in a different environment! Good programming habits preclude this kind of reliance on side-effects.

5.13 RECURSIVE FUNCTION CALLS

Recursive and nonrecursive function calls are handled in the same way by the LISP evaluator: the method described in Section 5.11 does not make any provision for recursive functions.

One serious problem may arise with unsound recursive definitions: the LISP evaluator may not be able to terminate. It is the user's responsibility to check his definitions with respect to termination.

Consider the following definition of the **MAD** function:

```
(DEFINE 'MAD '(X) '(CDR (MAD X)))
```

It is easy to see that the evaluation of any *MAD-form* will go on indefinitely. For any correct form f, of value f', the evaluation of (MAD f) will require the evaluation of the body (CDR (MAD X)) of MAD with X bound to f', which will require the evaluation of the argument (MAD X), which will ask for the evaluation of the body (CDR (MAD X)) with X bound to f', and so on! No value will ever be returned and CDR will never be applied.

The unsoundness of MAD's definition stems from the equation it implies:

```
(MAD X) = (CDR (MAD X))
```

If (MAD X) was ever defined, its value should, because of the above equality, have a CDR and therefore be a dotted pair (not an atom). But this dotted pair would be its own CDR–which is impossible, at least in pure LISP.

One sufficient condition of termination is that *something in the arguments of the recursive call(s) gets smaller than in the function parameters*. Clearly, MAD does not meet this criterion, since the argument of the recursive call (MAD X) is just MAD's parameter X. The question of termination plays a major role in the design of recursive functions.

5.14 SIMPLE GUIDELINES FOR DESIGNING RECURSIVE FUNCTIONS

There is no general method for designing functions, whether recursive or not. However, we believe that "thinking recursive" is often a good strategy for solving a problem: recursive solutions are usually easier to find, more natural, and more elegant than nonrecursive ones.

Here are some guidelines that may help the novice:

Rule 1. Think of the problem to be solved as that of transforming input into the desired result by means of a function f (the function being sought).

Rule 2. Find a decomposition of the input into smaller components, similar to the whole with respect to the structure, such that you can achieve Rule 3.

Rule 3. Supposing that each smaller part of the input could be transformed into a partial result, find a combination of LISP primitives or existing functions which, if they were applied in some appropriate manner to the partial results, would yield the desired global result.

Rule 4. Express the condition under which the input can be decomposed according to Rule 2. Try to solve the alternate case, using LISP primitives or existing functions. (This alternate case may split into several cases, called *base cases*.)

Rule 5. The function f should have the general form:

```
(DEFINE '      '(input)
        'COND  [(condition input)
                 (base case solution input)]
               [T (combination
                    (f   (select_part_1 input))
                                 ...
                    (f   (select_part_n input)))]))
```

Rule 6. Check f for termination–prove that when the condition on the input is not satisfied, then part 1 of the input is "smaller" than the input and part n of the input is "smaller" than the input.

Rule 7. Try to minimize the number of base cases.

Rule 8. Test the derived function using specific input data.

Many problems, difficult as well as easy ones, can be solved with the above strategy. Some cannot: for instance, this scheme will not build a system of mutually recursive functions–e.g., f contains a call to g, and g contains a call to f.

5.15 APPLICATION OF THE GUIDELINES TO THE DESIGN OF A SIMPLE TRANSLATOR

In Section 5.8 we considered algebraic expressions that were written in the so-called infix notation.

Suppose we wish to translate them into *Polish notation*. Polish notation is used by pocket calculators that do not supply parentheses. To perform "3 + 4" on these calculators, you enter 3, then 4, then +. A more complicated expression, such as (A * (B + C)) is represented by:

```
A B C + *
```

in Polish notation. No parentheses are necessary: the operand(s) always precede the operator. Since + and * are binary operators, there is only one possible interpretation: B and C are the operands of +; A and "B C +" are the operands of *.

Similarly, the expression (A + (COS B)) is represented by:

```
A B COS +
```

unambiguously.

Since LISP deals with atoms or dotted pairs, we shall adopt the convention of using lists to represent Polish objects, e.g.:

```
(A B C + *)      for    A B C + *,
(A B COS +)      for    A B COS +,
(A)              for    A
```

We need a function for attaching one list to another: this function is usually called **APPEND**. Given two lists u and v, it returns the list of all the elements of u followed by all the elements of v. For instance:

```
(APPEND '( B A)  '(C A B D))
```

evaluates to:

```
(A B A C A B D)
```

Writing a definition of APPEND is a good exercise for the reader. Here is a solution:

```
(DEFINE 'APPEND '(U V)
        '(COND [(LISTP U)
                (CONS (CAR U)
                      (APPEND (CDR U) V))]
               [T V]))
```

We can now solve the problem of translating infix notation (the usual notation for algebraic expressions) into Polish notation. Let us apply the rules of Section 5.14.

Rule 1. We call POLISH our function for transforming algebraic expressions into Polish lists. We call EXP the input expression.

Rule 2. We note that expressions can usually be broken into one or two subexpressions and a connecting operator. We already have LISP functions for selecting the components (see Section 5.8):

(OP EXP): the main operator of the expression

(LEFT EXP): the left operand (if the operator is binary)

(RIGHT EXP): the right operand.

Rule 3. Let us assume that (POLISH (LEFT EXP)) and (POLISH (RIGHT EXP)), respectively, yield the Polish form of EXP's left operand, if there is one, and the Polish form of EXP's right operand if EXP is non-atomic and can thus be decomposed. To get the Polish form of the whole expression, we need to build up one list containing, in this order:

all the elements of (POLISH (LEFT EXP))

all the elements of (POLISH (RIGHT EXP))

the main operator, (OP EXP)

by definition of the Polish notation.

The case where EXP has no left operand–that is, (LEFT EXP) evaluates to NIL–seems to require special treatment. However, this distinction will be superfluous if we let POLISH return NIL for NIL. Then the first argument of APPEND, which we shall use to append lists together, will eventually evaluate to NIL. Since (APPEND NIL V) always yields the value of V, this method will work.

Rule 4. The input EXP can be decomposed if EXP is not an atom or, equivalently, if EXP is a dotted pair.

The condition for the alternate case–the base case–is that EXP be atomic. According to our previous remark, we must treat NIL as a special atom:

if EXP is NIL, POLISH must return NIL

if EXP is any other atom, e.g., B, POLISH must return a list whose only element is this atom, e.g., the list (B); this list can be produced by (CONS EXP NIL).

Rule 5. Here is how we define POLISH:

```
(DEFINE 'POLISH
        '(EXP)
        '(COND [(ATOM EXP)
                (COND [EXP (CONS EXP NIL)]
                      [T   NIL])]
               [T
                (APPEND (POLISH (LEFT EXP))
                (APPEND (POLISH (RIGHT EXP)))
                        (CONS (OP EXP)
                              NIL))])))
```

Rule 6. When EXP is not atomic, it must have a left operand (NIL, eventually) and a right operand, provided that EXP is a well-formed expression. The number of atomic-symbol occurrences in each operand is smaller than the number of atomic-symbol occurrences in the whole expression by at least 1, because the main operator occurs once more in EXP. Hence, this function will always terminate if the input is a correct expression (in infix form).

There are, of course, other equivalent definitions of POLISH. Furthermore, our definition imposes a slight restriction on algebraic expressions: the symbol NIL should not be used! For instance, the expression (A + NIL) will be incorrectly translated into (A +) instead of (A NIL +). This restriction can be relieved with some other definition, e.g.:

```
(DEFINE 'POLISH
        '(EXP)
        '(COND [(ATOM EXP)
                (CONS EXP NIL)]
               [T
                (APPEND
                   (COND [(CDR (CDR EXP))
                          (POLISH (LEFT EXP))]
                         [T   NIL])
                   (APPEND (POLISH (RIGHT EXP))
                           (CONS (OP EXP) NIL))])))
```

The Polish notation in this section is called *postfix*, because it places each operator to the right of its operands. Conversely, the *prefix* Polish notation places the operator in front of its operands, i.e., (* A + B C) means (A * (B + C)). Very little change needs to be done to any of these definitions in order to obtain a translator from infix notation to prefix notation. We leave it as an exercise for the reader.

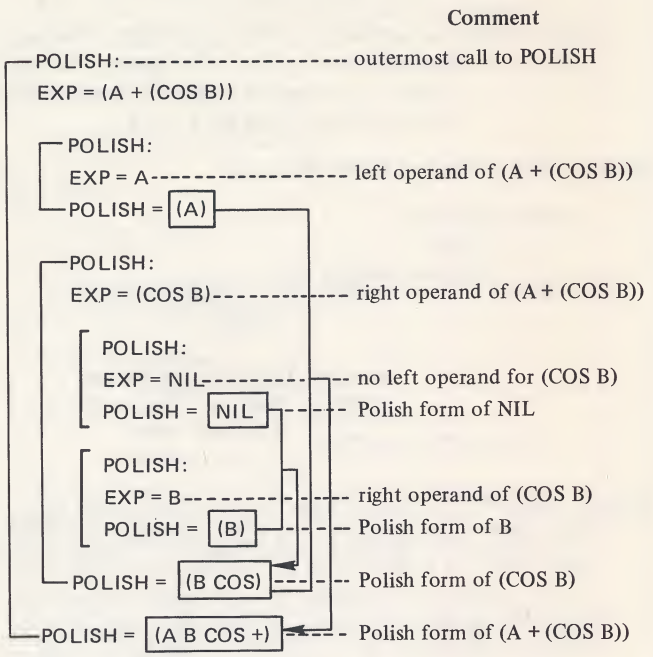
To understand recursion more fully, the reader should try to simulate evaluation of the form:

```
(POLISH '(A + (COS B)))
```

using our POLISH function and applying the evaluation process described in previous sections. You can then try it on any LISP system that provides a TRACE function. By typing something like:

```
(TRACE '(POLISH))
```

and then the form given above, a trace similar to the one below will be displayed:



5.16 ANOTHER EXAMPLE: SYMBOLIC DIFFERENTIATION

Many systems based on LISP—e.g., LISP-REDUCE, MACSYMA—are able to perform symbolic calculus and may be of great help to mathematicians and engineers.

We now want to construct a very simple differentiator working in a very limited context. It takes algebraic expressions in input. They contain +, *, and – as the only binary operators, and COS, SIN, and OPP as the only unary operators. (OPP means "opposite of": we need to distinguish the use of "–" in "–X" from its use in "A – B". The first "–" corresponds to the unary operator OPP, while the latter is the binary operator.) Here are the mathematical laws of differentiation we need to incorporate into our function:

$$
\begin{aligned}
D(u + v) &= Du + Dv \\
D(u - v) &= Du - Dv \\
D(u\,v) &= uDv + vDu \\
D(-u) &= -Du \\
D(\sin u) &= (\cos u)\,Du \\
D(\cos u) &= -(\sin u)\,Du
\end{aligned}
$$

To be consistent with the conventions used previously, we rewrite these laws:

```
(D (U + V))   =  ((D U) + (D V))
(D (U – V))   =  ((D U) – (D V))
(D (U * V))   =  ((U * (D V)) + (V * (D U)))
(D (OPP U))   =  (OPP (D U))
(D (SIN U))   =  ((COS U) * (D U))
(D (COS U))   =  (OPP ((SIN U) * (D U)))
```

We first define two elementary functions, MAKE1 and MAKE2, which build up an expression from an operator and one or two operands:


```
(DEFINE 'MAKE1
        '(OP RIGHT)
        '(CONS OP
               (CONS RIGHT NIL)))

(DEFINE 'MAKE2
        '(LEFT OP RIGHT)
        '(CONS LEFT
               (CONS OP
                     (CONS RIGHT NIL))))
```

See how close to the laws of differentiation the following LISP definition is (**DIFF** is the name of our function):

```
(DEFINE 'DIFF '(EXP)
  '(COND [(ATOM EXP)
          (MAKE1 'D EXP)]
         [(EQ (OP EXP) '+)
          (MAKE2 (DIFF (LEFT EXP))
                 '+
                 (DIFF (RIGHT EXP)))]
         [(EQ (OP EXP) '–)
          (MAKE2 (DIFF (LEFT EXP))
                 '–
                 (DIFF (RIGHT EXP)))]
         [(EQ (OP EXP) '*)
          (MAKE2 (MAKE2 (LEFT EXP)
                        '*
                        (DIFF (RIGHT EXP)))
                 '+
                 (MAKE2 (RIGHT EXP)
                        '*
                        (DIFF (LEFT EXP))))]
         [(EQ (OP EXP) 'OPP)
          (MAKE1 'OPP (DIFF EXP))]
         [(EQ (OP EXP) 'SIN)
          (MAKE2 (MAKE1 'COS (RIGHT EXP))
                 '*
                 (DIFF (RIGHT EXP)))]
         [(EQ (OP EXP) 'COS)
          (MAKE1 'OPP
              (MAKE2 (MAKE1 'SIN (RIGHT EXP))
                     '*
                     (DIFF (RIGHT EXP))))]
         [T
          (MAKE1 'D EXP)]))
```

The functions OP, LEFT, and RIGHT were defined in Section 5.8. Note the last clause of the COND—when the main operator is unknown, DIFF simply returns the expression with D in front. For instance, (DIFF '(TG (X + Y))) evaluates to (D (TG (X + Y))), since DIFF does not know the differential of TG.

However, this simple differentiator behaves nicely on expressions such as:

```
((SIN (X * Y)) + Z)
```

for which the following differential will be computed:

```
((COS ((X * (D Y)) + (Y * (D X)))) + (D Z))
```

DIFF always terminates on algebraic expressions given in infix form: the reason is exactly the same as for POLISH (see the application of Rule 6 of Section 5.15).

6. MORE PROGRAMMING CONSTRUCTS AND CONCEPTS

The concepts and constructs presented so far are essential and sufficient for writing meaningful functions, at least in theory. More expertise may be acquired through reading well-written LISP programs. However, real programs contain additional constructs and concepts. The purpose of this chapter is to inform the reader of their existence and utility in preparation for practical LISP programming.

We strongly discourage the reader from trying to learn or use subsequent material until you have gained full mastery of the basic principles developed in Chapters 1 to 5. You might then never be able to take advantage of the powerful and simple ideas that make up the originality of LISP!

6.1 DESIGNING INTERACTIVE PROGRAMS: PRINT, READ, TERPRI

LISP systems are usually interactive. User-defined functions can be made interactive, too.

The form (**PRINT** form) will output the value of the argument form at your terminal (or another device–line-printer, disk, etc.). If your LISP system supports strings, you can have your function issue some message, e.g., "How old are you?" by incorporating the form (PRINT "How old are you?").

Similarly, **READ** can be used within a function to get information from the user of the function. Typically, the form:

```
(SETQ ANSWER (READ))
```

has the effect of waiting until some S-expression is entered on the terminal and then binding the atom ANSWER to this S-expression (which is actually the value of (READ)). Note that READ takes no argument.

TERPRI is a function of no argument that issues a carriage return so that subsequent messages will appear on the next line.

There may be many other input/output functions available on your system and installation. Some LISPs include graphic or acoustic functions, as well as functions for controlling mechanical devices.

6.2 SEQUENTIAL PROGRAMMING: PROGN OR IMPLICIT PROGN

It may be desirable to specify a sequence of forms to be evaluated, rather than just one form. This effect can be achieved by means of the **PROGN** function, which takes a variable number of arguments. The syntax is:

(PROGN $form_1, \ldots, form_n$)

All the arguments from $form_1$ to $form_n$ are evaluated in that order; the value of the PROGN form is the value of the last one, $form_n$. If none of the forms $form_1, \ldots, form_{n-1}$ is able to change the LISP environment (i.e., bind some atom) or interact with the external world, then there is no point in using PROGN, since $form_n$ will have the same effect and value.

Sequential programming is thus closely related to the idea of side-effect. A *side-effect* is a change of the LISP environment or external world.

Most LISPs support implicit PROGNs. For instance, the HANOI definition of Section 2.3 contains an implicit PROGN. It is equivalent to:

```
(DEFINE 'HANOI '(N SOURCE INT DEST)
        '(COND [(EQUAL N 0) NIL]
               [T (PROGN (HANOI (SUB1 N) SOURCE DEST INT)
                         (MOVE_ONE_DISK)
                         (HANOI (SUB1 N) INT SOURCE DEST))]))
```

Tolerant LISP systems permit the body of a function to be a sequence of forms rather than just one form. PROGN is of little use with such systems.

6.3 PROG, RETURN, AND GOTO

The general syntax of *PROG-forms* is:

(PROG ($local_1 \ldots local_k$)
$stat_1$
...
$stat_n$)

where:

($local_1 \ldots local_k$) is used to set up a local environment, for the time of the evaluation of the PROG-form. Each element of this list is either an atom or a pair of the form (atom init). The atom specifies a local variable, which should be bound to the value of the form init. (When no initial value is specified, NIL is assumed.) The local variables should bear distinct names, just like parameters in a function definition. Local variables should be compared to function parameters and init-forms to arguments in a function call. In fact, the first steps of the evaluation of the PROG-form are precisely steps 1, 2, 3 of the evaluation process described in Section 5.11, up to the renaming suggested by this comparison. Step 5 of this process also takes place at the end, just before leaving the PROG-form.

$stat_1, \ldots, stat_n$ are either labels or forms (usually called *statements*), including GOTO-statements or RETURN-statements but excluding atomic form. They constitute the body of the PROG-form and are evaluated in sequence, except when a GOTO-statement or RETURN-statement has been encountered.

A label is an atom: it is not evaluated but serves as a marker of some point in the PROG-form. Labels within a PROG-form must be distinct.

A GOTO-statement is a pair of the form (GOTO lab), where lab is an atom. This atom should be a label in the same PROG-form or in some embedding PROG-form or even (for some LISPs only) a label in some PROG-form that is currently active–that is, a PROG-form for which there still exists an entry on the stack. In the first case–the usual case–the effect of (GOTO lab) is simply to disrupt the sequential course of evaluation of the statements and start over from the point marked by lab. (Note that lab is not evaluated.) In the latter case, the stack is POPped until an

active PROG-form containing lab as a label is found. Each time the stack is POPped, the corresponding PROG-form or function call is exited, which means that the local variables or function parameters are restored.

A RETURN-form is the normal exit from a PROG-form. It takes one argument. When (RETURN form) is encountered, form is evaluated first; then the PROG-form is exited, with the local variables restored (that is, rebound to the values they had prior to the evaluation of the PROG-form), and the value of form is returned as value of the PROG-form.

6.4 LOOPS, RECURSION, AND ITERATIVE CONSTRUCTS

A typical use of PROG is the implementation of a loop, e.g.:

```
(PROG   ((L initial))
  TEST  (COND
          [L (PROGN  first_form
                     ...
                     last_form
                     (SETQ L (CDR L))
                     (GOTO TEST))]
          [T (RETURN some_result)]))
```

in which the local variable L is bound to the value of initial upon entering PROG; TEST is a label. At the beginning of the loop, L is tested: if it evaluates to NIL, the PROG-form is exited and the value of some_result is returned; otherwise, the forms first_form, . . . , last_form are evaluated in sequence–assuming they do not contain any RETURN or GOTO statements. At the end of the loop, L is reset to its own CDR and the evaluation starts again from TEST, the beginning of the loop.

Assuming that first_form, . . . , last_form do not alter L and that L is initially a list of *n* elements, this loop will be evaluated exactly *n* times.

This kind of program fragment is called *iterative*, in contrast to recursive functions. It is possible to achieve the same effect without a GOTO statement, by defining a recursive function as follows:

```
(DEFINE  'LOOP
         '(L)
         '(COND
            [L PROGN  first_form
                      ...
                      last_form
                      (LOOP (CDR L))]
            [T  some_result] ))
```

and using the form:

```
(LOOP initial)
```

instead of the PROG-form. (The PROGN is even superfluous with LISPs that support implicit PROGN.)

The main advantage of iterative implementation of the loop over the recursive one is that the GOTO-statement does not consume stack memory, whereas each evaluation of the recursive call (LOOP (CDR L)) pushes something onto the stack. (This is not true of the VLISP interpreter, which is able to detect an iteration here and will simulate a GOTO!)

Finally, the PROG-form is a relatively obsolete feature when powerful iterative constructs are available. In INTERLISP for example, the form:

```
(PROGN (FOR L ON initial DO first_form . . . last_form)
       some_result)
```

will achieve the same loop. Other colorful constructs include:

```
(FIND TOMATO IN my_basket SUCHTHAT you_like_it)
(REPEAT treatment UNTIL prisoner_talks)
(FOR COUNTRIES IN the_west JOIN the_armies)
```

which almost speak for themselves! (TOMATO and COUNTRIES are atoms; my_basket, you_like_it, treatment, prisoner_talks, the_west, the_ armies stand for LISP forms, FIND, IN, SUCHTHAT, REPEAT, UNTIL, FOR, JOIN are INTERLISP keywords.)

6.5 EVAL EVALUATES ITS ARGUMENT A SECOND TIME

EVAL is a function of one argument. The value of:

```
(EVAL form)
```

is by definition *the value of the value of form*. The form can be any LISP form, provided that the value of form is itself a LISP form! *Through EVAL, a LISP program can build a LISP form and then evaluate this LISP form.* Although this feature may seem esoteric at first glance, it is invaluable in practice.

6.6 A SIMPLE PROBLEM THAT CLASSICAL PROGRAMMING LANGUAGES CANNOT SOLVE

Let us consider the problem of drawing the graphs of numerical functions whose algebraic definitions are entered at run-time by the user. Here is the schema of a typical LISP program that solves this problem:

```
(DEFINE 'DRAW_GRAPH '( )
  (PROG (FORMULA X)
    ASK (PRINT ' "Please type a formula.")
        (TERPRI)
        (SETQ FORMULA (READ))
        (COND
          [(EQ FORMULA 'STOP) (RETURN NIL)]
          [bad_formula (GOTO ASK)]
          [T  (LISPIFY FORMULA)]
        (SETQ X 0.)
        (START_DRAWING_FROM X (EVAL FORMULA))
   LOOP (SETQ X (+  X  0.1))
        (DRAW_TO X (EVAL FORMULA))
        (COND [(EQUAL X 10.) (GOTO ASK)]
              [T (GOTO LOOP)])))
```

This program first asks the user to enter a formula. A correct formula should express the ordinate as a mathematical function of the abscissa X, e.g.:

```
((COS X) + (3 * X))
```

If the user wants to exit, she types STOP. Otherwise, the formula is checked (see bad_formula). When the formula is

acceptable (e.g., the formula above), it is translated into an equivalent LISP form, e.g.:

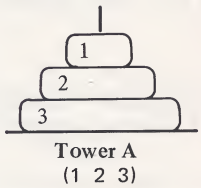
```
(+ (COS X) (* 3 X))
```

by LISPIFY, a function as simple as POLISH (see Section 5.15).

Then starts the drawing. We have assumed the existence of two graphic functions: given the coordinates x and y of some screen location, START_DRAWING_FROM initializes the drawing from this point; DRAW_TO draws a line segment from the current position to the point (x, y). For simplicity, we let x go from 0 to 10 by steps of 0.1; obviously, a more flexible design is possible. *This program cannot be written in FORTRAN, ALGOL, PL/I, Pascal, BASIC, or ADA. (It can be written in APL.)*

6.7 AN IMPLEMENTATION OF MOVE_ONE_DISK BASED ON EVAL

Let us go back to the Towers of HANOI, introduced in Section 2.2. Before we can write the MOVE_ONE_DISK program, we must choose a representation of the towers. We number the disks from 1 to N (the total number of disks) by order of increasing size. A tower is then a list of numbers. We list disks from the top to the bottom of each tower, so that the CAR function can easily "grasp" the topmost disk. Here is a tower A and its representation in the initial state, for N = 3.



To represent all three towers A, B, and C in any state, we let the values of the atoms A, B, and C be the representations of towers A, B, and C, respectively:

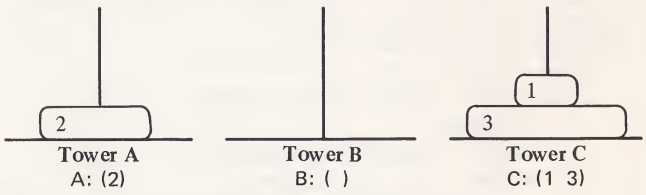


FIGURE 6-1. Representation of the Towers

Recall the function of MOVE_ONE_DISK: Move the topmost disk from the tower whose name is the value of SOURCE to the top of the tower whose name is the value of DEST (SOURCE and DEST are parameters of the calling function HANOI). Hence the following definition:

```
(DEFINE 'MOVE_ONE_DISK '( )
        '(PROGN
             (SET DEST
                  (CONS (CAR (EVAL SOURCE))
                        (EVAL DEST)))
             (SET SOURCE
                  (CDR (EVAL SOURCE)))
             display_move_graphically))
```


It is important to realize that MOVE_ONE_DISK does not reset the HANOI parameters DEST and SOURCE: it resets the atoms, i.e., tower names, which are the values of DEST and SOURCE, since SET is used instead of SETQ (see Section 4.8).

To make this point clear, let us assume the following bindings (as pictured in Figure 6-1):

```
SOURCE: C    INT: B    DEST: A
```

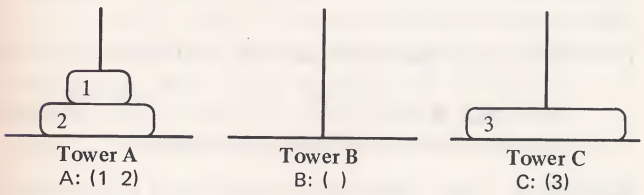
and:

```
C: (1 3)    B: ( )    A: (2)
```

Upon entering MOVE_ONE_DISK:

- SOURCE evaluates to C; (EVAL SOURCE) evaluates to (1 3)
- (CAR (EVAL SOURCE)) evaluates to 1, which is C's topmost disk
- (EVAL DEST) evaluates to the value of A, which is (2)
- The first SET resets A to (1 2)
- The second SET resets C to (3)

which completes the move and yields the state:



Note that A, B, and C play the role of *global variables:* they are not parameters of any function. If they were parameters of MOVE_ONE_DISK, the side-effect would be undone upon leaving MOVE_ONE_DISK, and nothing would ever move!

The following forms are sufficient to set up the initial towers and run HANOI for N = 3:

```
(SETQ N 3)
(SETQ A '(1 2 3))
(SETQ B NIL)
(SETQ C NIL)
(HANOI 3 'A 'B 'C)
```

6.8 PROPERTY LISTS AND KNOWLEDGE REPRESENTATION

We have already seen two different kinds of information that can be attached to a literal atom (other than NIL or T) in the symbol table:

- A *value* (by means of SETQ or as a result of function-call evaluation)
- A *definition* (by means of the DEFINE function)

Conceptually, values and definitions are all we need. *Property lists* are a third kind of information that can be attached to literal atoms. This new feature does not add expressive power to LISP but is of tremendous practical interest. The main application is *data bases*.

The property list of an atom generalizes the concept of value. A property list is *a multiple value.* Each element of this multiple value has a name, usually called the *property name* or *property*.

Consider, for example, an individual named STEVEN. STEVEN's age is 23. He is a male. We can represent this information by attaching the *value* 23 under the *property* AGE to the *atom* STEVEN and attaching the *value* MALE under the *property* SEX to the same *atom* STEVEN. AGE and SEX are property names: any literal atom can serve as a property name. Thus:

- The *value* of the *property* AGE of the *atom* STEVEN is 23
- The *value* of the *property* SEX of the *atom* STEVEN is MALE

Suppose that we now want to express the fact that STEVEN has two friends, TIMOTHY and LUCY. We simply attach the value (TIMOTHY LUCY) to the atom STEVEN under the property FRIENDS. If we wish to say that TIMOTHY is 17, we can attach the value 17 to TIMOTHY under the property AGE.

Each LISP system has its own set of primitive functions for dealing with property lists. Here are the two functions we need, borrowed from the INTERLISP dialect:

(PUTPROP atom propname propval) attaches the value of propval to the value of atom under the property name that is the value of propname. Both atom and propname should evaluate to literal atoms.

(GETPROP atom propname) returns the S-expression stored under the property propname's value of the atom atom's value. NIL is returned if nothing has been stored under this property. Both atom and propname should evaluate to literal atoms.

To set up this story about STEVEN, TIMOTHY, and LUCY, it suffices to type:

```
(PUTPROP 'STEVEN 'AGE 23)
(PUTPROP 'STEVEN 'SEX 'MALE)
(PUTPROP 'STEVEN 'FRIENDS '(TIMOTHY LUCY))
(PUTPROP 'TIMOTHY 'AGE 17)
```

Making reasonable assumptions, we can also type:

```
(PUTPROP 'TIMOTHY 'SEX 'MALE)
(PUTPROP 'LUCY 'SEX 'FEMALE)
```

Then the form:

```
(GETPROP 'STEVEN 'FRIENDS)
```

will evaluate to the list:

```
(TIMOTHY LUCY)
```

and:

```
(GETPROP 'TIMOTHY 'AGE)
```

will evaluate to 17.

Property lists are thus an adequate tool for knowledge representation, under the form of *semantic networks*. The semantic network we just built is drawn in Figure 6-2.

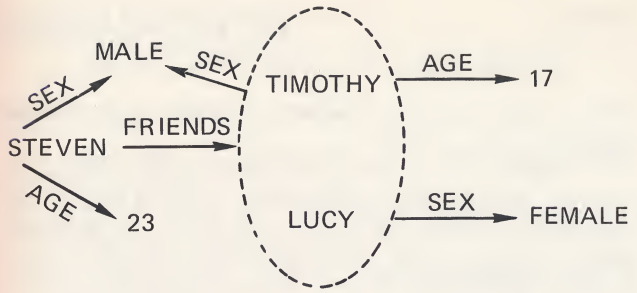


FIGURE 6-2. A Simple Semantic Network

Property lists are also a convenient means of attaching LISP forms to atoms using PUTPROP; these forms can be retrieved by GETPROP and evaluated with EVAL (see Section 6.5).

6.9 MAPPING FUNCTIONS

Consider the simple function defined by:

```
(DEFINE 'SQUARE '(N) '(* N N))
```

It computes the square of a given number N.

Suppose that, given a list of numbers, we want to get the list of their squares in the same order. We could define a recursive or iterative function for doing that. It is simpler to use a mapping function such as MAPCAR, which is predefined in most LISP dialects. For instance:

```
(MAPCAR '(1 3 5 10) 'SQUARE)
```

evaluates to the list of squares:

```
(1 9 25 100)
```

In general, if *list* evaluates to the list $(e_1\ e_2 \ldots e_n)$ and *funct* evaluates to some function name (or LAMBDA-expression) f, the effect of MAPCAR is shown by the following diagram:

$$\begin{array}{rl} \text{list:} & (e_1 \quad e_2 \quad \ldots \quad e_i \quad \ldots \quad e_n) \\ & \quad\downarrow f \quad\downarrow f \qquad\downarrow f \qquad\downarrow f \\ \text{(MARCAR list funct):} & (r_1 \quad r_2 \quad \ldots \quad r_i \quad \ldots \quad r_n) \end{array}$$

in which r_i, for $1 \leq i \leq n$, is the result of applying f to e_i, i.e., the value of $(f\ e_i)$.

Understanding mapping functions leads to a better understanding of iterative constructs, because these are often implemented (via macros) in terms of mapping functions (and PROGs). Nowadays, iterative constructs (see Section 6.4) often supersede mapping functions.

6.10 ANONYMOUS FUNCTIONS

In the example in Section 6.9, we could avoid defining a SQUARE function—and thus naming it—by using the anonymous LAMBDA-expression:

```
(LAMBDA (N) (* N N))
```

instead of SQUARE in the form:

```
(MAPCAR '(1 3 5 10) 'SQUARE)
```

We type:

```
(MAPCAR '(1 3 5 10) '(LAMBDA (N) (* N N))
```

This is a typical example of a suitable use of anonymous functions.

However, there is a problem with recursive functions. How can we make them anonymous, since the body of the definition contains a call to the function itself? Some LISPs offer a solution. VLISP tells you to use the keyword *SELF* in place of the function name in each recursive call. For example, the classical definition of *FACTORIAL* being:

```
(DEFINE 'FACTORIAL '(N)
        '(COND [(EQUAL N 0) 1]
               [T (* N (FACTORIAL (-N 1)))]))
```

the anonymous equivalent of FACTORIAL is:

```
(LAMBDA (N) (COND [(EQUAL N 0) 1]
                  [T (* N (SELF (- N 1)))]))
```

6.11 LACUNAE

We hope to have conveyed the essential ideas of LISP in this Handy Guide. However, we have, either willingly or unwillingly, left out quite a lot.

Among these omissions are **macros**, which in essence permit extension of the LISP language; **selective evaluation** of arguments (NLAMBDA), variable numbers of arguments (SPREAD), which are only used in top-level functions to alleviate typing; and the **EVALQUOTE mode**, which is one of the most confusing features for beginners, and we have not even alluded to **coroutines**, **spaghetti stacks**, and the like, the **funarg problem** (FUNCTION), or **list-destructive** functions (RPLACA, RPLACD, CONC, NCONC, . . .), which deserve attention after a certain level of understanding of LISP has been reached.

To avoid confusion, we have deliberately opted to explain a single concept when two or more are available. In the important case of function-call evaluation, for instance, we have chosen the INTERLISP conception: *the CAR of a function call is not evaluated, whether it is an atom or a LAMBDA-expression*, although some LISP systems do evaluate this CAR under certain conditions.

BIBLIOGRAPHY

The following are good textbooks:

1. John Allen, *Anatomy of LISP*, McGraw-Hill, New York, 1979.
2. Laurent Siklossy, *Let's Talk LISP*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
3. Clark Weissman, *LISP 1.5 Primer*, Dickenson (Wadsworth), Belmont, CA, 1967.
4. Patrick H. Winston & Berthold K. P. Horn, *LISP*, Addison-Wesley, Reading, MA, 1981.

These are reference books dealing with specific versions of LISP:

5. James R. Meehan, Ed., *New UCI LISP Manual*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1979.
6. David Moon, *MACLISP Reference Manual*, Laboratory of Computer Science, MIT, Cambridge, MA, 1974 (Revised 1978).
7. L. Quam & W. Diffie, *Stanford LISP 1.6 Manual*, SAIL, Stanford University, Stanford, CA, 1972.
8. Warren Teitelman, *INTERLISP Reference Manual*, Xerox Corporation, Palo Alto Research Center, Palo Alto, CA, 1974 (Revised 1978).
9. Daniel Weinreb & David Moon, *LISP Machine Manual*, AI Laboratory, MIT, Cambridge, MA, 1978 (Revised 1979).

The following are references on functional programming:

10. John Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, Vol. *21*, No. 8, pp. 613-641, August 1978.
11. Peter Henderson, *Functional Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

INDEX

Alfred Handy Guides

Practical, economical, and concise
Alfred Handy Guides tell you what you need to know quickly and easily—without a lot of reading!

Perfect for today's fast-moving adult on the run, they fit anywhere—in pocket, purse, gadget bag, guitar case. Always where you need them!

The Alfred Handy Guide Series to Computers

How to Buy a Personal Computer
How to Buy a Word Processor
How to Use VisiCalc/SuperCalc
Understanding APL
Understanding Artificial Intelligence
Understanding Atari Graphics
Understanding BASIC
Understanding COBOL
Understanding Data Base Management
Understanding FORTRAN
Understanding LISP
Understanding Pascal

Other Alfred Handy Guide Series

Cooking
Health
Music
Photography

Look for new titles and new series.
For more information:

Alfred Publishing Co., Inc.
P.O. Box 5964
15335 Morrison St.
Sherman Oaks, CA 91413

ISBN 0-88284-219-6